

---

HP 64000 Logic Development System

**HP-UX Hosted  
Cross Assembler/  
Linker User Definable**

**Operating Manual**



HP Part No. 64851-97000

Printed in U.S.A.

June 1989

Edition 2



---

## Certification and Warranty

### **Certification**

Hewlett-Packard Company certifies that this product met its published specifications at the time of shipment from the factory. Hewlett-Packard further certifies that its calibration measurements are traceable to the United States National Bureau of Standards, to the extent allowed by the Bureau's calibration facility, and to the calibration facilities of other International Standards Organization members.

### **Warranty**

This Hewlett-Packard system product is warranted against defects in materials and workmanship for a period of 90 days from date of installation. During the warranty period, HP will, at its option, either repair or replace products which prove to be defective.

Warranty service of this product will be performed at Buyer's facility at no charge within HP service travel areas. Outside HP service travel areas, warranty service will be performed at Buyer's facility only upon HP's prior agreement and Buyer shall pay HP's round trip travel expenses. In all other cases, products must be returned to a service facility designated by HP.

For products returned to HP for warranty service, Buyer shall prepay shipping charges to HP and HP shall pay shipping charges to return the product to Buyer. However, Buyer shall pay all shipping charges, duties, and taxes for products returned to HP from another country. HP warrants that its software and firmware designated by HP for use with an instrument will execute its programming instructions when properly installed on that instrument. HP does not warrant that the operation of the instrument, or software, or firmware will be uninterrupted or error free.

### **Limitation of Warranty**

The foregoing warranty shall not apply to defects resulting from improper or inadequate maintenance by Buyer, Buyer-supplied software or interfacing, unauthorized modification or misuse, operation outside of the environment specifications for the product, or improper site preparation or maintenance.

**No other warranty is expressed or implied. HP specifically disclaims the implied warranties of merchantability and fitness for a particular purpose.**

### **Exclusive Remedies**

**The remedies provided herein are buyer's sole and exclusive remedies. HP shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory.**

Product maintenance agreements and other customer assistance agreements are available for Hewlett-Packard products.

For any assistance, contact your nearest Hewlett-Packard Sales and Service Office.

---

## Notice

**Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.**

Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

© Copyright 1984, 1985, 1989 Hewlett-Packard Company.

This document contains proprietary information, which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this document is subject to change without notice.

UNIX is a registered trademark of AT&T.

Torx is a registered trademark of Camcar Division of Textron, Inc.

**Logic Systems Division  
8245 North Union Boulevard  
Colorado Springs, CO 80920, U.S.A.**

---

## Printing History

New editions are complete revisions of the manual. The date on the title page changes only when a new edition is published.

A software code may be printed before the date; this indicates the version level of the software product at the time the manual was issued. Many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one to one correspondence between product updates and manual revisions.

Edition 1	<b>64851-90906 ,September 1984</b>
Edition 2	<b>64851-97000, June 1989</b>

## Using This Manual

---

This manual describes how to use the HP 64851S User Definable Assembler in a HP-UX environment. Create your custom assembler on an HP 64000 development station. After it has been created, upload the assembler to the host mainframe. At this point, the assembler can be used in the same way as any other assembler in the HP-UX environment. Follow the instructions in the first eight chapters to create the custom assembler and linker, then follow the uploading instructions in Chapter 9.

---

**Note**

Be certain to read the CAUTION on page 7-1 and "Sample Code Defining 8080 Processor" on page 7

---

---

## Notes



# Contents

---

<b>1</b>	<b>General Information (User Definable Assembler/Linker)</b>	
	Introduction . . . . .	1-1
	Assembler Operation . . . . .	1-3
	What The User Must Define . . . . .	1-4
<b>2</b>	<b>Programming Rules</b>	
	Introduction . . . . .	2-1
	User Definable Assembler Structure . . . . .	2-2
	Defining the Processor . . . . .	2-2
	Defining Relocatable Code Generation Formats . . . . .	2-3
	Internal Constants . . . . .	2-4
	Predefined Symbols . . . . .	2-4
	Instruction Group . . . . .	2-5
	Defining the Instruction Set (INSTR_DEF) . . . . .	2-5
	Parsing the Instruction Set (INSTR_SET) . . . . .	2-6
<b>3</b>	<b>Assembler Commands, Symbols, Instructions, and Conventions</b>	
	Introduction . . . . .	3-1
	Assembler Directive . . . . .	3-1
	Assembler Setup Commands . . . . .	3-2
	Predefined Symbols . . . . .	3-5
	Pseudo Instructions . . . . .	3-7
	Assembler Instructions . . . . .	3-8
	Conventions . . . . .	3-11
<b>4</b>	<b>Assembler Subroutines</b>	
	Introduction . . . . .	4-1
	Subroutines And Examples . . . . .	4-1
	Column Pointers . . . . .	4-1
	ADD_LABEL . . . . .	4-3
	CHECK_AUTO_DEC . . . . .	4-3
	CHECK_AUTO_INC . . . . .	4-3

CHECK_COMMA	4-4
CHECK_DELIMITER	4-4
CHECK_EOL	4-5
CHECK_EXPR_ERROR	4-5
CHECK_PASS1_ERROR	4-5
COUNTER_UPDATE	4-7
ERROR	4-7
EVEN n	4-9
EXPRESSION	4-9
EXPRESSION_2	4-10
FIND_DELIMITER	4-11
GEN_CODE	4-11
GET_ASCII_BYTE	4-12
GET_OPCODE	4-12
GET_PROG_COUNTER	4-13
GET_START_CHAR	4-13
GET_STOP_CHAR	4-13
GET_SYMBOL	4-14
GET_TOKEN	4-15
NOT_DUPLICATE	4-17
PRINT_LOCATION	4-17
SAVE_ERROR	4-17
SAVE_WARNING	4-17
SCAN_REAL	4-17
UPDATE_LABEL	4-19
WARNING	4-19

## 5 Creating An Assembler

Introduction	5-1
Summary Of The Assembler Source Code Building Process for 8080 Processor	5-2
Assembler Setup Commands	5-2
Defining and Parsing the Instruction Set (INSTR_DEF & INSTR_SET)	5-4
Tracing The User Defined Assembler Execution Sequence	5-6

## 6 Linker General Information

Introduction	6-1
Linker Operation	6-1

<b>7</b>	<b>Linker Programming Rules</b>	
	Linker Structure . . . . .	7-1
	Linker Setup Commands . . . . .	7-3
	Processor Definition . . . . .	7-4
	Sample Code Defining 8080 Processor . . . . .	7-4
	Define Entry Points For Relocatable Routines . . . . .	7-6
	Linker Instructions . . . . .	7-6
	Predefined Symbols . . . . .	7-9
	Relocatable Format Routines . . . . .	7-10
<b>8</b>	<b>Creating The Linker</b>	
	Introduction . . . . .	8-1
	Tracing The User Defined Linker Execution Sequence . . . . .	8-3
<b>9</b>	<b>Uploading To The Mainframe</b>	
	Introduction . . . . .	9-1
	Uploading Assembler Tables . . . . .	9-1
	Uploading Linker Tables . . . . .	9-1
<b>A</b>	<b>User Defined Assembler Code for 8080 Processor</b>	
<b>B</b>	<b>User Defined Linker Code for 8080 Processor</b>	
<b>C</b>	<b>Summary of Assembler Subroutines</b>	
<b>D</b>	<b>Relocatable and Absolute File Formats</b>	
	Nam Record (record Type = 1) . . . . .	D-2
	Glb Record (record Type = 2) . . . . .	D-3
	Dbl Record (record Type = 3) . . . . .	D-4
	Ext Record (record Type = 4) . . . . .	D-5
	End Record (record Type = 5) . . . . .	D-6
	Absolute File . . . . .	D-7

# Illustrations

---

Figure 1-1. User Definable Assembler/Linker Overview . . .	1-2
Figure 1-2. Assembler Functions . . . . .	1-3
Figure 2-1. Assembler Building Process . . . . .	2-2
Figure 4-1. Forward Referenced Symbol Code Gen. Chart . .	4-6
Figure 5-1. Creating the Assembler . . . . .	5-2
Figure 5-2 Example of TRACE 2 Output . . . . .	5-6
Figure 6-1. Linker Module Functions . . . . .	6-2
Figure 7-1. Linker Building Process . . . . .	7-2
Figure 8-1. Creating the Linker . . . . .	8-2



## General Information (User Definable Assembler/Linker)

---

### Introduction

An assembler translates mnemonic source code into object code that will execute on a specific processor. The user definable assembler/linker permits the instruction set and instruction format of any processor to be defined in a source program by the user. In addition, it can be used to modify source type HP Model 64000 Assemblers by adding or changing instructions. Assembler code for the Model 64000 is modular and changes can also be made by merging code in appropriate places.

### Note



---

The user definable assembler/linker cannot be used to modify existing ABSOLUTE assembler files.

---

The assembler and linker both have two modules:

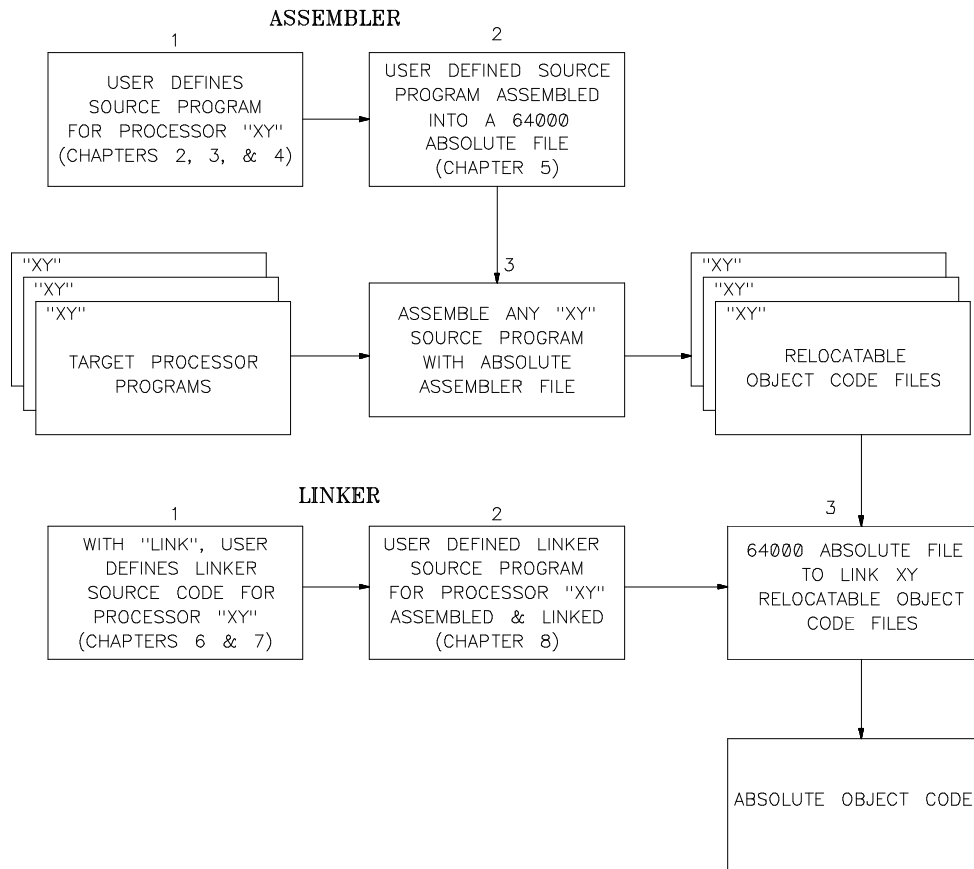
1. The basic assembler module that is part of the Model 64000 operating system and cannot be modified by the user.
2. The user definable assembler module.
3. The basic linker module, which is also part of the Model 64000 operating system and cannot be modified by the user.
4. The user definable linker module.

Figure 1-1 illustrates how the user definable assembler and linker are created and then used with target system programs for the user processor.

**Note**



Refer to the *Assembler/Linker Reference Manual* for details on the basic assembler and linker modules. This manual supplement will only describe the user definable assembler and linker modules.



**Figure 1-1. User Definable Assembler/Linker Overview**

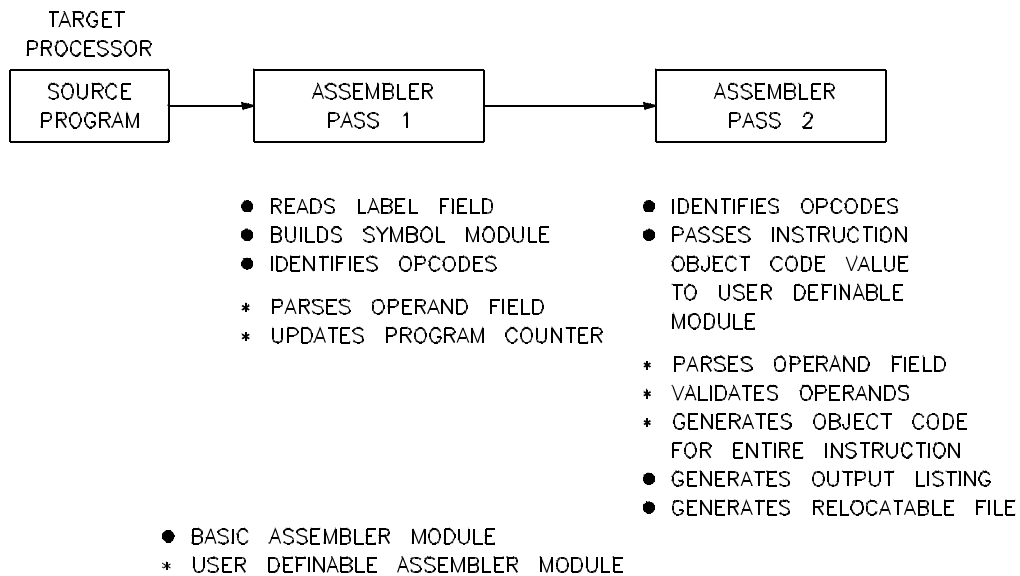
**1-2 General Information**

## Assembler Operation

HP 64000 Assemblers include a pass 1 and a pass 2. The same code is used to generate both passes. Primary functions in pass 1 are building the symbol table and updating the program counter. To build the symbol table, labels and operands are identified and stored by names and addresses or labels. Object code is generated in pass 2, based on the symbol table.

The programmer implements the functions the user definable assembler must perform with a set of subroutines. These subroutines will be explained in Chapter 4 of this supplement. The functions performed by the basic assembler module and user definable assembler module are shown in figure 1-2.

The user defines the instruction set and predefined registers and symbols. The standard set of pseudo instructions can be used as is, redefined, or extra pseudo instructions peculiar to the user's assembly language can be added. The assembler also includes a symbol table building method that is mostly transparent to the user.



**Figure 1-2. Assembler Functions**



---

## What The User Must Define

To define an assembler program, the user must provide the following information.

1. Identify all predefined symbols for registers, stack pointers, condition codes, etc. for the target processor.
2. Divide the instruction set into separate groups of instructions that are parsed in the same way.
3. Identify the machine code corresponding to the "unalterable" part of each instruction (opcode).
4. Define the parsing rules for each instruction group.



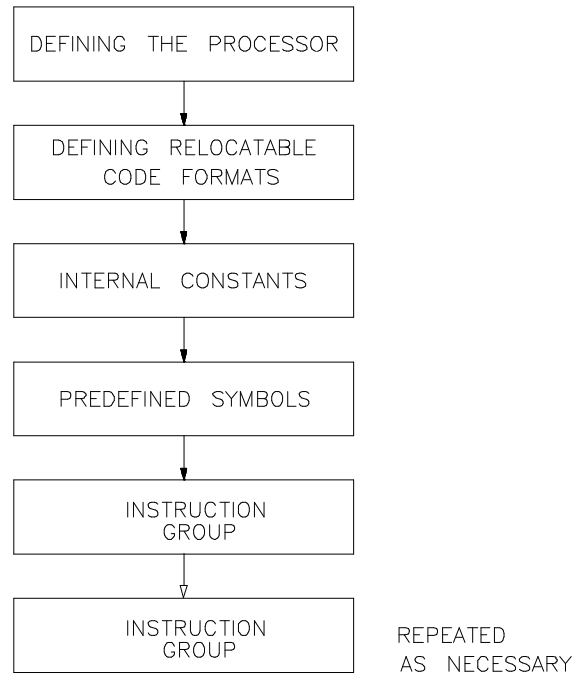
# Programming Rules

---

## Introduction

This chapter will explain the tasks that must be completed before user definable assembler code can be written. The functional block diagram in figure 2-1 illustrates the assembler building process. Each block corresponds to a paragraph title.

1. The user processor must be defined, including all predefined symbols for its language.
2. Instructions must be divided in groups that can be parsed in the same way and then defined in machine code (INSTR\_DEF).
3. The parsing rules for each instruction group in step b must be specified. This defines how to handle the instruction set (INSTR\_SET).



**Figure 2-1. Assembler Building Process**

---

## User Definable Assembler Structure

### Defining the Processor

In this first section of user definable code, setup commands define the basic parameters of the user processor. For example, assembler directive, word size, address size, assembler list title, print field size, linker file identifier, constants, registers, status words, and stack pointers. The 8080 processor will be used in the examples shown in this manual. Some information about the processor is included here. For more details, refer to the *8080/8085 Assembler Supplement*.

In the following examples, some of the user definable assembler setup commands are illustrated. Chapter 3 discusses all the setup

### 2-2 Programming Rules

commands. The setup commands can be in any order desired by the programmer, except for the assembler directive, which must be the FIRST setup command.

Example:

```
ASSEMBLER "8080"           ;Defines the processor.
WORDSIZE = 8              ;Defines the word size.
ADDRESS_BASE = 8          ;Specifies the program counter increment.
TITLE = "8080"            ;Title for the assembler list.
LOC_SIZE = 4              ;Four characters in the print field for the location
                           ;counter.
LINK_FILE L8080 : XX      ;Specifies linker file. XX is the USERID
                           ;(1 to 6 characters).
PC_16                     ;Only the lower 16 bits of the program counter are used.
```

### Defining Relocatable Code Generation Formats

The relocatable code formats must also be defined at this time with the RELOC\_FMT setup command. This command can be located anywhere in the group of setup commands. The command is used as follows.

```
RELOC_FMT <name> , SIZE = <n>
```

where:

<name>

is used in conjunction with GEN\_CODE to identify the relocatable addressing mode. GEN\_CODE will be explained in a later paragraph, Parsing the Instruction Set.

SIZE = <n>

defines the variable size being parsed (n= 1 to 32 bits).

Examples:

```
RELOC_FMT HIGH_LOW, SIZE = 16 ;Relocate 16 bits.
RELOC_FMT LOW_HIGH, SIZE = 16 ;Relocate and swap bytes.
RELOC_FMT LOW_BYTE, SIZE = 8  ;Low byte, no error check.
RELOC_FMT HIGH_BYTE, SIZE = 8 ;High byte, no error check.
RELOC_FMT LOW_CHECK, SIZE = 8 ;Low byte, check for >256.
RELOC_FMT REL_8, SIZE = 8     ;Plus minus 128.
RELOC_FMT PC_REL, SIZE = 8    ;-126, +129
```

## Internal Constants

Assembler internal constants are used for the programmers convenience. In the examples below, the temporary registers TEMP1, TEMP2, and TEMP3 are assigned a new name under CONSTANTS to aid in program documentation. There are 40 temporary registers available to the programmer (TEMP1 to 40).

Examples:

```
CONSTANTS
HIGH_FLAG = TEMP 1           ;Used as a flag if HIGH keyword is found.
COUNT = TEMP2              ;Used as a temporary count.
MEM_CHECK = TEMP3           ;Used to check memory reference on MOV instructions
END
```

## Predefined Symbols

Predefined symbols can be defined to represent registers, status words, stack pointers, etc.

Examples:

```
OBJ.
CODE
0006          SYMBOLS = REGISTER           ;Defines the TYPE and
                                           ;VALUE assigned to the
0007          A = 7                       ;symbols. REGISTER is
0000          B = 0                       ;TYPE 6. Symbol C has
0001          C = 1                       ;a VALUE of 1.
.            D = 2
.            E = 3
.            H = 4
.            L = 5
.            M = 6
END

SYMBOLS = STATUS
    PSW = 6
END
SYMBOLS = STACK
    SP = 6
END
SYMBOLS = ADDR_OPER
    HIGH = 1
    LOW = 0
END
```

During assembler operation, values assigned to the symbols will be used by the assembler subroutines.

## 2-4 Programming Rules

---

## Instruction Group

### Defining the Instruction Set (INSTR\_DEF)

The user must now divide the instruction set into separate groups of instructions that are parsed in the same way. Depending on the processor being defined, common parsing rules could include instruction format, data format, addressing modes, etc. This allows all instructions within a group to be handled in the same manner, which simplifies assembler operation.

The definition of each group must start with INSTR\_DEF. This is followed by each instruction and its object code format. It is used as follows:

```
INSTR_DEF [OPERAND = X] [SPACES]
```

initiates section of code where the instruction mnemonics are equated to their respective machine codes. OPERAND= X and SPACES are optional parameters and are specified on the same line. X is the number of operands in a source statement to be cross referenced. OPERAND= 0 turns off cross referencing for the instruction group. DEFAULT: if OPERAND is not specified, all operands in the source statement are cross referenced.

SPACES is a key word used by the cross-reference generator to develop cross-reference tables. The key word "SPACES" indicates to the cross-reference generator that spaces are permitted in the operand field for the target processor. Note that SPACES must be used if it applies to the target processor. Each INSTR\_DEF section is followed by an INSTR\_SET section.

Example:

```
INSTR_DEF  OPERAND=0          ;Starts instruction set
                                ;definition section for
                                ;no operand instructions.

                                CMC = 3FH
                                RIM = 20H
                                .
                                .
                                .
                                HLT = 76H
```

## Parsing the Instruction Set (INSTR\_SET)

This next section defines the parsing rules that will perform the object code conversion for the user processor. It must start with INSTR\_SET and terminate with the DONE instruction. The following example illustrates the basic structure. Each instruction group made up of INSTR\_DEF and INSTR\_SET must terminate with an END instruction. An example assembler source program with details on exactly how code is written is provided in Chapter 5. Chapter 4 explains the user definable assembler subroutines.

Example:

```
INSTR_DEF OPERAND=0

    CMC = 03FH
    ..
    ..
INSTR_SET                                ;Starts source code parsing section.

    GEN_CODE ABS 8, OBJECT_CODE
    DONE                                  ;Return to basic assembler module.
END                                       ;Must terminate instruction group.

INSTR_DEF                                ;Starts next instruction group definition section.
    ..
    ..
    Code
    ..
    DONE
END
```

This continues until each instruction group for the processor is defined.

The print formats and code generating rules are defined with the GEN\_CODE subroutine. For absolute code this is accomplished by setting up GEN\_CODE parameters that define the size of the generated code in bits (8 or 16) and the predefined operand that contains the binary code to be generated. The GEN\_CODE subroutine is explained in detail in Chapter 4.

Example:

```
GEN_CODE ABS 8, OBJECT_CODE              ;The code size is 8 bits.
                                          ;The predefined symbol
                                          ;OBJECT_CODE will contain
                                          ;the bit pattern to be
                                          ;generated.
```

## 2-6 Programming Rules

For relocatable code, the GEN\_CODE subroutine has a different format and is used with the RELOC\_FMT setup command described earlier. It has the following form.

GEN\_CODE < name> , VALUE[SPACE]

or (either VALUE or BOTH must be specified)

GEN\_CODE < name> , BOTH[SPACE]

where:

< name>	is used in conjunction with GEN_CODE to identify the relocatable addressing mode.
VALUE	uses the contents of the predefined symbols VALUE and relocation TYPE to generate code.
[SPACE]	inserts a space in the object code field of the assembler listing.
BOTH	uses the contents of the predefined symbols VALUE, relocation TYPE, and OBJECT_CODE to generate code.

---

## Notes





## Assembler Commands, Symbols, Instructions, and Conventions

---

### Introduction

This chapter first explains the assembler directive and the setup commands needed to define the user processor. Predefined symbols are identified next, followed by pseudo and assembler instructions. An explanation of the conventions used completes the chapter.

---

### Assembler Directive

In Chapter 2, under "Defining the Processor", brief examples show how a processor is defined. In defining a processor, the first statement must be the assembler setup command ASSEMBLER, followed by the assembler directive in quotes.

Example:

```
ASSEMBLER "8080"
```

After the processor is defined, target system source programs must always begin with the assembler directive.

```
"8080"  
source code  
"  
"  
END
```

---

## Assembler Setup Commands

Use the setup commands to define basic parameters such as assembler directive, word size, address size, constants, registers, status words, and stack pointers. Except for the assembler directive, which must be first, the order of the setup commands is left to the programmer's discretion.

ADDRESS\_BASE            defines the process address mode; i.e.,  
= nn                      word or byte. Defaults to eight bits.

ASSEMBLER                defines the assembler directive for the  
"< name> "                user processor.

LINK\_FILE                allows the user to define the linker  
module to be used during a target system  
source program link operation. If an HP  
system linker absolute module exists on  
the Model 64000, it can be used,  
providing no additional formats or no  
system linker is available, a user  
definable linker module must be  
defined. An example of the LINK\_FILE  
setup command for the system linker  
module and the user definable linker  
module follows:

(system absolute linker module) LINK\_FILE I8085\_Z80 : HP  
(user defined absolute linker module) LINK\_FILE L8080 :  
USERID

### Note



---

The user linker name (L8080 here) can be any legal file name. The system linker module uses a lower case I identifier and is stored under USERID HP.

---

LOC\_SIZE = n            sets up the size of the print field for the  
location counter (n= 1 to 8 characters).  
DEFAULT: four characters.

DOUBLE\_ADDRESS defines 32-bit addresses to be passed to the linker.

PC\_16 indicates only the lower 16 bits of the program counter will be incremented.

RELOC\_FMT < name> , SIZE = < n>

< name> is used in conjunction with GEN\_CODE to identify the relocatable addressing mode. The GEN\_CODE subroutine is explained in Chapter 4.

SIZE = < n> defines the variable size being parsed (n= 1 to 32 bits).

RENAME\_PSEUDO allows the user to rename the pseudo provided by the Model 64000 system. It has the following format:

RENAME\_PSEUDO < new name of pseudo> = < pseudo number>

Example:

RENAME\_PSEUDO ORIGIN = 1

The list of pseudos and their associated pseudo number follow:

**Note**



---

The IF pseudo cannot be renamed.

---

PSEUDO	PSEUDO NUMBER
ORG	1
PROG	2
DATA	3
COMN	4
EQU	5
EXT, EXTERNAL	6
GLB, GLOBAL	7
LIST	8
SPC	9
NAME	10
REPT	11
SKIP	12
TITLE	13
MASK	14
END	15
WARN	16
NOWARN	17
NOLIST	18
EXPAND	19
HEX	20
DEC, DECIMAL	21
OCT, OCTAL	22
BIN, BINARY	23
ASC, ASCII	24
INCLUDE	25
TRACE	26
REAL	27
SET	28

**SYMBOLS = < name>** defines user definable types. See TYPE under Predefined Symbols.

**TITLE = "< string> "** defines the header line on the assembler list output.

**WORD\_SIZE = nnn** defines the processor word size. Allowable range is 8 to 128 bits. DEFAULT: eight bits.

### 3-4 Commands, Symbols, Instructions, & Conventions

---

## Predefined Symbols

The following symbols are reserved. They have special meaning to the basic assembler module and cannot be redefined by the user.

### Note



---

All variables and registers are 32 bits long.

---

ACCUMULATOR	working register.
AUTO_DEC_COUNT	set by CHECK_AUTO_DEC and used by EXPRESSION.
AUTO_INC_COUNT	set by CHECK_AUTO_INC and used by EXPRESSION.
CHARACTER	used by CHECK_DELIMITER, GET_START_CHAR and GET_STOP_CHAR to return the character found.
CLASS	returned by GET_TOKEN with an indicator of the token type found:  0= Numeric constant 1= Undefined 2= String constant 3= Operator 4= Delimiter 5= Upper case variable 6= Undefined 7= Lower case variable 8= Undefined 9= End of line-no tokens in string 10= Decimal constant with E notation

*EXT_ID_NUMB	variable returned EXPRESSION and GET_SYMBOL with an external variable identification number assigned by the assembler.
*EXT_OFFSET	variable returned by EXPRESSION and GET_SYMBOL with the value of the offset to be added to an external operand at link time.
*For more information, refer to EXPRESSION and GET_SYMBOL subroutines in Chapter 4.	
INSTR_RESET	variable reset to 0 at the beginning of each instruction.
OBJECT_CODE	register used to pass the object code to the code generating routine.
PROGRAM_COUNTER	variable identifying the current TYPE of code. See TYPE 0 through 3.
RESULT	variable containing the value of the TOKEN returned by GET_TOKEN.
SAVE_PTR	pointer set by EXPRESSION to save the position of the STOP pointer at the time EXPRESSION was invoked.
START	pointer used by subroutines to control the scanning function.
STOP	pointer used by subroutines to control the scanning function.
TOKEN_ERROR	set by GET_TOKEN when an error is found.
TYPE	variable containing the type of an evaluated expression.

### 3-6 Commands, Symbols, Instructions, & Conventions

0= absolute  
1= program relocatable  
2= data relocatable  
3= common relocatable  
4= external reference  
5= equated to external  
6> user definable types (see SYMBOLS).

VALUE                      variable containing the value of an expression.



---

## Pseudo Instructions

Pseudo instructions are used by most assemblers to provide for special functions that are not part of the basic instruction set. They are used to define storage space, equate variable names to specific values, identify labels to variable names, etc. In some cases nonexecutable code is generated for assembler pseudo instructions, while in other cases, such as listing control and constant definition, no code is generated.

All of the standard pseudo instructions explained in the *Assembler/Linker Reference Manual* are available to the user. In addition, these standard instructions can be renamed as explained earlier in this chapter, under "Assembler Setup Commands", `RENAME_PSEUDO`.

The `TRACE` pseudo enables the user to examine execution of user definable assembler code. For more details and an example, refer to "Tracing the User Definable Assembler", in Chapter 5.

---

## Assembler Instructions

Use the following assembler instructions in the INSTR-SET section to implement the instruction group parsing rules. All arithmetic is performed in two's complement, 32 bits wide. Be certain to read the next section, "Conventions".

ADD operand	add the contents of "operand" to the contents of the ACCUMULATOR. The result remains in the ACCUMULATOR.
AND operand	logically ANDs the "operand" with the contents of the ACCUMULATOR. The result remains in the ACCUMULATOR.  ACCUMULATOR <-- ACCUMULATOR AND operand
CALL label	transfers program execution to the subroutine at the address specified by label.
DECREMENT operand	decrements the "operand" by one.
DONE	terminates INSTR_SET code and transfers control to the basic assembler module.
END	indicates the end of an assembler module. Each module must be terminated by an END instruction.
GOTO label	transfers program execution to the address specified by label.
IF operand1 "condition" operand2 THEN instruction	compares operand1 with operand2 according to the specified "condition." If "condition" is true, instruction is executed. If not, control is transferred to



the instruction immediately after the IF instruction.

"condition" can be:

- > greater than
- > equal to or greater than
- < less than
- < less than or equal to
- = equal to
- < > not equal to



**Note**



---

All comparisons are unsigned.

---

INCREMENT operand increments the contents of "operand" by one. operand <-- operand + 1

LOAD operand loads the ACCUMULATOR with the contents of "operand."

ACCUMULATOR <-- operand

NOP no operation.

OR operand logically ORs the contents of "operand" with the contents of the ACCUMULATOR. The result remains in the ACCUMULATOR.

ACCUMULATOR <--  
ACCUMULATOR OR operand

RETURN n transfers program control to the "nth" instruction after the CALL instruction. If n is omitted, a return 1 is executed by default.

SHIFT_LEFT n	shifts the ACCUMULATOR contents n bits to the left. Zeros are filled in. $0 <= n <= 32$ .
SHIFT_RIGHT n	shifts the ACCUMULATOR contents n bits to the right. Zeros are filled in. $0 <= n <= 32$ .
STORE operand	stores the contents of the ACCUMULATOR in "operand."  operand <-- ACCUMULATOR
STORE_0 operand	clears the contents of "operand."  operand <-- 0
STORE_1 operand	sets bit 0 of "operand" and clears all other bits.  operand <-- 1
SUBTRACT operand	subtracts "operand" contents from ACCUMULATOR contents and stores results in ACCUMULATOR.  ACCUMULATOR <-- ACCUMULATOR - operand
TWOS_ COMPLEMENT	calculates the two's complement of ACCUMULATOR contents.  ACCUMULATOR <-- ACCUMULATOR + 1

### 3-10 Commands, Symbols, Instructions, & Conventions

---

## Conventions

Observe the following conventions when programming.

Auto decrement	automatic decrement function is represented by a trailing minus sign; e.g., An-.
Auto increment	automatic increment function is represented by a trailing plus sign; e.g., An+ .
Blank line	blank lines are ignored by the assembler modules.
Comment field	begins with a semicolon.
Comment line	if a semicolon is in the first column, the entire line is treated as a comment.
Delimiters	legal delimiters are: space ; , \$ : @ ! % # ' & ? . \ / ~ { } or end of line.
End of line	a blank, semicolon, or actual end of line are valid end of line indicators.
Hex notation	the first digit in hexadecimal notation must be a numeral 0 through 9. The suffix H must also be present. For example, F8 in hexadecimal is 0F8H.
Indexing	specified by brackets, [ ]; e.g., [Rn].
Label	identifies a statement. Every label is unique within a source program. A label can be up to 110 characters long, but only the first 15 are used for identification.

---

## Notes



## Assembler Subroutines

---

### Introduction

This chapter explains all the assembler subroutines and illustrates their operation with one or more examples where appropriate. The assembler subroutines are arranged alphabetically. For quick reference, an alphabetical summary of all the subroutines appears in Appendix C.

Back in Chapter 2, how to define and implement a user instruction set was briefly described (see `INSTR_DEF` and `INSTR_SET`). By the end of this current chapter, the user will have seen all the assembler subroutines. At this point, the building process has been explained. Chapter 5 shows how to create the assembler program; it also lists a sample 8080 program if further clarification is needed.

---

### Subroutines And Examples

#### Note



---

When program control passes from the basic assembler module to the user definable module, the `START` and `STOP` pointers are positioned at the first character in the operand field if the delimiter is a blank. If another delimiter is present, both pointers will be at the delimiter.

---

#### Column Pointers

There are two column pointers (`START` and `STOP`) not visible to the programmer. Their column location can be identified with the

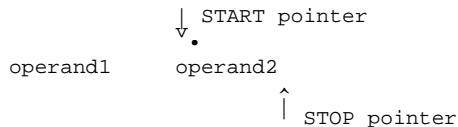
TRACE pseudo instruction. Refer to "Tracing the User Definable Assembler" in Chapter 5 for an example. These column pointers are initialized to the start of the operand field by the user definable assembler and are used by the subroutines. In most cases, the subroutines called will move the pointers as required; however, they can be moved by the programmer using the assembler instructions. In the subroutine examples that follow, the pointer positions are shown to clarify the subroutine explanation. There is an additional pointer, SAVE\_PTR, used with the EXPRESSION subroutine. SAVE\_PTR saves the initial position of the STOP pointer. It is useful for flagging errors in expression VALUES and/or TYPES. An example of how the pointers are moved follows.

Example:

There are two operands in the source line. The first operand has been evaluated by the EXPRESSION subroutine and the second operand is to be evaluated next. The STOP pointer is at the first space after operand 1 and there are one or more spaces between the operands.



The subroutine GET\_TOKEN is used to get the next token in the source statement (operand 2). GET\_TOKEN begins at the STOP pointer and skips to the first nonblank column. The START pointer is placed at the beginning of the token and the STOP pointer is placed at the first column past the token.



To use the subroutine EXPRESSION on operand 2, the STOP pointer must be at the beginning of operand 2. The STOP pointer is moved with the LOAD and STORE instructions. LOAD START loads the column value of the START pointer into the accumulator. STORE STOP stores the contents of the accumulator in the STOP pointer.

## 4-2 Assembler Subroutines

```

LOAD START      operand1      operand2
STORE STOP
                ^
                | START pointer
                ^
                | STOP pointer

```

EXPRESSION can now evaluate operand 2.

### ADD\_LABEL

Puts a label found in the operand field in the symbol table during pass 1. Stores VALUE and TYPE. A return 1 is executed if there is no label. A return 2 is executed if a label is found. This allows the user to insert symbols in the symbol table in addition to the standard symbol table building performed by the assembler.



### CHECK\_AUTO\_DEC

Checks for auto decrement in the form of a trailing operator(s). For example, A- or A--; the - sign(s) represents the auto decrement operator(s). AUTO\_DEC\_COUNT is set to the number of trailing operators found. In the example A--, it is set to 2. If no operators are found, it is set to 0.

Both CHECK\_AUTO\_DEC and the next subroutine, CHECK\_AUTO\_INC, are used in conjunction with the EXPRESSION subroutine. If an expression can legally end in - or +, then these subroutines should be used.

Example:

```

CHECK_AUTO_DEC  R10-
EXPRESSION
                ^
                | STOP pointer after EXPRESSION
                | is invoked
R10-

```

Note, if the subroutine is not called before EXPRESSION, then EXPRESSION will flag the - sign as an error.

### CHECK\_AUTO\_INC

Checks for auto increment in the form of a trailing operator(s). For example, B+ or B++ ; the + sign represents the auto increment operator(s). AUTO\_INC\_COUNT is set to the number of trailing operators found. If no operators are found, it is set to 0.

## CHECK\_COMMA

Checks the token at the STOP pointer for a comma. If a comma is not present, a return 1 is executed and the STOP pointer is not changed. If a comma is found, a return 2 is executed and the STOP pointer is incremented by one.

Examples:

```
↓ STOP pointer before CHECK_COMMA is invoked.
MVI A:LABEL
↑ STOP pointer after return 1.

↓ STOP pointer before CHECK_COMMA is invoked.
MVI A, LABEL
↑ STOP pointer after return 2.
```

## CHECK\_DELIMITER

Checks for a delimiter at the position indicated by the STOP pointer. If an end of line is found (blank, semicolon, or actual end of line), a return 1 is executed. If the character found is not a legal delimiter, a return 2 is executed and the STOP pointer is not altered. If a legal delimiter is found, the STOP pointer is incremented, the delimiter is stored in CHARACTER, and a return 3 is executed. Legal delimiters were listed under "Conventions", in Chapter 3.

Examples:

```
↓ STOP pointer before CHECK_DELIMITER is invoked.
MVI
↑ STOP pointer after return 1.

↓ STOP pointer before CHECK_DELIMITER is invoked.
MVI A>LABEL
↑ STOP pointer after return 2.

↓ STOP pointer before CHECK_DELIMITER is invoked.
MVI A, LABEL
↑ STOP pointer after return 3.

CHARACTER now contains ", "
```

### 4-4 Assembler Subroutines



## CHECK\_EOL

Checks for a valid end of line; i.e., a blank, a semicolon, or the actual end of line. A return 1 is executed if a valid end of line is found. A return 2 is executed if no valid end of line is found. The STOP pointer is not incremented after return 1 or return 2.

Example:

```
      ↓ STOP pointer before CHECK_EOL is invoked.  
MVI A,LABEL  
      ↑ STOP pointer after return 1 or return 2.
```

## CHECK\_EXPR\_ERROR

After the EXPRESSION handler is called, CHECK\_EXPR\_ERROR can determine if an error has been flagged by EXPRESSION. If an error is found, a return 1 is executed. If no error is found, a return 2 is executed.

Example:

```
EXPRESSION          ;Evaluate expression.  
CHECK_EXPR_ERROR   ;Check for error.  
GOTO ERROR_EX      ;Error subroutine.  
LOAD VALUE  
..  
..  
..  
ETC
```

## CHECK\_PASS1\_ERROR

A problem arises when a symbol is used in the operand field before it is defined in the symbol table (forward reference). The missing information can introduce an error in the program counter. For example, if the subroutine EXPRESSION is used in pass 1 and a symbol is not defined, the quantities in VALUE and TYPE will not be defined. If the same symbol is defined later, the subroutine EXPRESSION will return the appropriate VALUE and TYPE in pass 2, but the program counter will differ between the two passes, and a different number of bytes of code will be generated. Two error checking routines are included in the user definable assembler to warn the programmer of these oversights.

In either pass 1 or pass 2, if a symbol was not defined when the routine is invoked, the CHECK\_PASS1\_ERROR routine returns program control to the instruction immediately following the routine call. If the symbol was defined in pass 1, program control is passed to the second instruction following the routine call.

When a syntax error is found by the EXPRESSION subroutine, the CHECK\_EXPR\_ERROR subroutine allows the assembler to stop parsing. Using both error subroutines differentiates between pass 1 errors and syntax errors. The usual sequence of steps and associated code is shown in the next example.

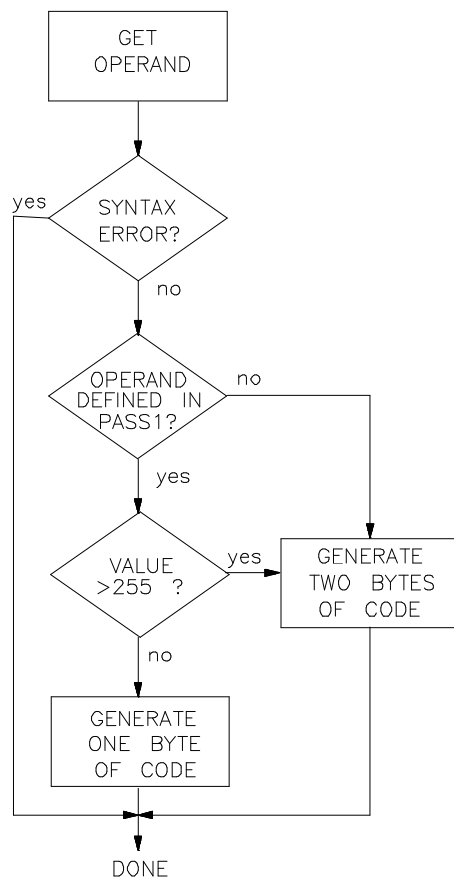


Figure 4-1. Forward Referenced Symbol Code Gen. Chart

#### 4-6 Assembler Subroutines

**Example:**

```
EXPRESSION ;Get operand.
CHECK_EXPR_ERROR ;Was there a syntax error?
DONE ;Yes, terminate
CHECK_PASS1_ERROR ;Was there a pass 1 error?
GOTO OUTPUT_TWO ;Yes- two bytes address.
IF VALUE >255 GOTO OUTPUT_TWO
GEN_CODE ABS 8 VALUE ;Generate one byte of code.
DONE
OUTPUT_TWO
GEN_CODE ABS 16 VALUE ;Generate two bytes of code.
DONE
ERROR_ROUTINE
ERROR DE_ERR ;Definition error.
DONE
```

**COUNTER\_UPDATE** Increments the program counter by the amount contained in VALUE.

```
program_counter <-- program_counter + VALUE
```

**ERROR** An error message is displayed from the following list. For example, ERROR IO\_ERR.

AS_ERR	ASCII string
CL_ERR	Conditional label
DE_ERR	Definition error
DS_ERR	Duplicate symbol
DZ_ERR	Division by zero
EE_ERR	Expected end of line
EG_ERR	External global
EO_ERR	External overflow
ES_ERR	Expanded source
ET_ERR	Expression type
IC_ERR	Illegal constant
ID_ERR	Invalid delimiter
IE_ERR	Illegal expression
IO_ERR	Invalid operand
IS_ERR	Illegal symbol
IP_ERR	Illegal parameter
LR_ERR	Legal range
MC_ERR	Macro conditional
MD_ERR	Macro definition
ML_ERR	Macro label
MM_ERR	Missing MEND
MO_ERR	Missing operator
MP_ERR	Mismatched parenthesis
MS_ERR	Macro symbol
NI_ERR	Nested includes
OS_ERR	Operand syntax
PC_ERR	Parameter call
PE_ERR	Parameter error
RC_ERR	Repeat call
RM_ERR	Repeat macro
SE_ERR	Stack error
TR_ERR	Text replacement
UC_ERR	Undefined conditional
UE_ERR	Unexpected end of line
UO_ERR	Undefined opcode
UP_ERR	Undefined parameter
US_ERR	Undefined symbol

#### 4-8 Assembler Subroutines

**EVEN n** Increments the program counter to an even word boundary if it is set to an odd value. "n" sets the program counter to the next value with "n" trailing zeros.

#### EXECUTE\_OPCODE

Assumes that the STOP pointer is positioned at the start of a user defined opcode. The subroutine looks up the opcode, initializes OBJECT\_CODE, and branches to the proper format in the user defined machine code. This occurs just as if the opcode was the first one encountered in the source statement.

Examples:

```

      ↓ STOP pointer before EXECUTE_OPCODE is invoked.
OPCODE MVI A,LABEL
      ↑ STOP pointer after EXECUTE_OPCODE is invoked.

      ↓ STOP pointer before EXECUTIVE_OPCODE is invoked.
MVI A,LABEL ;Error, not a valid user defined opcode.
      ↑ STOP pointer after EXECUTE_OPCODE is invoked.
```

**EXPRESSION** Evaluates expressions in the operand field and flags syntax errors in these expressions. Before the subroutine is invoked, the STOP pointer is at the beginning of the expression. After EXPRESSION is invoked, the STOP pointer moves to the next delimiter. The initial position of the STOP pointer is saved in SAVE\_PTR as shown in the following example. The SAVE\_PTR pointer is useful for flagging errors in expression VALUES and/or TYPES.

Example:

```

      ↓ STOP pointer before EXPRESSION is invoked.
MVI A,LABEL ;Error, not a valid user defined opcode.
      ↑ STOP pointer after EXPRESSION is invoked.
      ↑ SAVE_PTR after EXPRESSION is invoked.
```

EXPRESSION returns two predefined variables: VALUE, which contains the value of the expression and TYPE, which contains the type of the expression. A list of the various expression types follows.

```
TYPE
0   Absolute
1   Program relocatable
2   Data relocatable
3   Common relocatable
4   External reference
5   Equated to external
>6  User definable
```

The EXPRESSION subroutine sets up the following parameters used by the linker.

EXT\_OFFSET - value of the offset to an external variable such as in: EXT SAM, SAM1 EQU SAM+ 10. SAM1 is external and has an offset of 10.

EXT\_ID\_NUMB- identification number assigned to each external symbol.

## **EXPRESSION\_2**

Performs exactly like EXPRESSION except for the following two cases:

1. When an open parenthesis is encountered immediately following an operand token in an expression, the evaluation will be cleanly terminated and the VALUE (and other parameters) of the expression up to that point will be returned. The STOP pointer will be left pointing at the open parenthesis.
2. An initial '\*' in an expression is considered to be identical with '\$' (current location counter). Note that while '\$' can occur anywhere in the expression, '\*' must occur as the first token in the expression in order not to be mistaken for its use as the multiplication operator.

This version of EXPRESSION is primarily useful in evaluating operand fields where an index register can be enclosed in parenthesis.

**FIND\_DELIMITER** Finds the next delimiter in the current operand field.

Example:

```
↓ STOP pointer before FIND_DELIMITER is invoked.  
MVI A, LABEL  
↑ STOP pointer after FIND_DELIMITER is invoked.
```

**GEN\_CODE** Generates absolute or relocatable object code according to the parameters chosen. The program counter is incremented after the code is generated by the amount specified in the GEN\_CODE instruction.

Absolute code is generated with:

```
GEN_CODE ABS <n> , <operand> [SPACE]
```

where:

<n> is the code size in bits (8 or 16)

<operand> contains the bit pattern to be generated; e.g., VALUE, OBJECT\_CODE, etc.

[SPACE] inserts a space in the object code field of the assembler listing.

Relocatable code is generated with:

```
GEN_CODE <name>, VALUE [SPACE]
```

or (either VALUE or BOTH must be specified)

```
GEN_CODE <name>, BOTH [SPACE]
```

where:

<name> is used in conjunction with GEN\_CODE to identify the relocatable addressing mode.

VALUE uses the contents of the predefined symbols VALUE and relocation TYPE to generate code.

BOTH uses the contents of the predefined symbols VALUE, relocation TYPE, and OBJECT\_CODE to generate code.

The default instruction is GEN\_CODE < name> , VALUE.

## GET\_ASCII\_BYTE

Retrieves one ASCII character from an ASCII string within quotation marks. The START pointer must be at the left quote and the STOP pointer must be at the character after the right quote. A return 1 is executed if an end-of-string is found. A return 2 is executed when a valid character is found. The character is stored in the ACCUMULATOR.

### Note



---

The number of characters in the string is equal to: STOP pointer minus START pointer, minus 2. GET\_TOKEN should be called prior to GET\_ASCII\_BYTE. Then the START and STOP pointers will be set so this subroutine will operate properly.

---

### Example:

```
↓ START pointer before GET_ASCII_BYTE is invoked.
DB "ASCII string"
...
...
↑ STOP pointer before GET_ASCII_BYTE is invoked.
LOOP_BACK
    GET_ASCII_BYTE                ;Get character.
    GOTO END_OF_STRING            ;End-of-string found
    GEN_CODE ABS 8 ACCUMULATOR SPACE ;Generate one byte.
    GOTO LOOP_BACK                ;Get another character.
END_OF_STRING
...
...
```

## GET\_OPCODE

Checks for an opcode. Starts checking at the token indicated by the STOP pointer. Used for multiple opcodes. The value of opcode is placed in VALUE.

### Example:

CMA,RLC,DAA

After parsing the CMA instruction, we need to return to the instruction code parsing module to check for the RLC and the DAA instructions. This is achieved by calling GET\_OPCODE after each instruction mnemonic is parsed.



**GET\_PROG\_COUNTER** Returns the value of the user's source code program counter in the ACCUMULATOR.

ACCUMULATOR <-- PROGRAM\_COUNTER

Example: (Note this is a Z80 instruction)

↓ STOP pointer before EXPRESSION is invoked.

```
JR LABEL
```

↑ STOP pointer after EXPRESSION is invoked.

```
EXPRESSION           ;Get LABEL address.
GET_PROG_COUNTER     ;Get value of PC from ACCUMULATOR.
SUBTRACT VALUE       ;Offset = PC - LABEL.
```

**GET\_START\_CHAR** Retrieves the character indicated by the START pointer. A return 1 is executed if an end of line is found. A return 2 is executed when a valid character is found and placed in CHARACTER. The START pointer is then incremented by one.

Examples:

↓ START pointer before GET\_START\_CHAR is invoked.

```
MVI
```

↑ START pointer after return 1.

In this case, the START pointer was at an end of line.

↓ START pointer before GET\_START\_CHAR is invoked.

```
MVI A,LABEL,H
```

↑ START pointer after return 2

CHARACTER now contains ","

**GET\_STOP\_CHAR** Retrieves the character indicated by the STOP pointer. A return 1 is executed if an end of line is found. A return 2 is executed if a valid character is found. The character is stored in CHARACTER and the STOP pointer is incremented by one.

### Examples:

↓ STOP pointer before GET\_STOP\_CHAR is invoked.  
MVI  
↑ STOP pointer after return 1.

↓ STOP pointer before GET\_STOP\_CHAR is invoked.  
MVI A,LABEL,H  
↑ STOP pointer after return 2.

CHARACTER now contains "H"

### GET\_SYMBOL

Checks for a symbol. Starts checking at the token indicated by the STOP pointer. A return 1 is executed if the token is not a symbol (label or user defined symbol) and the STOP pointer remains unchanged. A return 2 is executed if the symbol is not in the symbol table and the STOP pointer remains unchanged. A return 3 is executed if the symbol was identified. VALUE and TYPE contain the value and type of the identified symbol.

#### Example:

↓ STOP pointer before GET\_SYMBOL is invoked.  
MVI A,LABEL  
↑ STOP pointer after return 3.

If the symbol (A) is identified, the routine will set up the following parameters.

VALUE: the value assigned to the symbol.

TYPE: the type assigned to the symbol.

If the symbol is external, the routine will set up the following parameters.

EXT\_ID\_NUMB: identification number assigned to each external/global symbol.

EXT\_OFFSET: value of the program counter offset; e.g.,  
used in program counter +  
displacement addressing modes (JP  
\$+ EXT).

**Note**



---

If a return 2 is executed in pass 1, the same return will be taken in pass 2 even though the symbol may have been defined for pass 2.

---

**GET\_TOKEN**

Gets the next token in the source statement. The subroutine begins at the position of the STOP pointer and skips to the first nonblank column. A token is identified in the source statement with the START pointer at the beginning and the STOP pointer at the first column past the token. Does a return 1 with CLASS containing the class of the token and RESULT containing the value of the token if the token is a numeric constant (CLASS= 0). A numeric constant starts with a digit and ends with one of the following characters to define the constant base: B- binary constant, H- hexadecimal constant, or O or Q- octal constant. If no character is present, a decimal constant is assumed.

```
CLASS
0   Numeric constant
1   Undefined
2   String constant
3   Operator
4   Delimiter
5   Upper case variable
6   Undefined
7   Lower case variable
8   Undefined
9   End of line- no tokens in the string.
10  Decimal constant with E notation.
```

Examples:

```
Class 0      0FFH
  ^
  | START pointer
  ^
  | STOP pointer

Class 2      "ABCD"
  ^
  | START pointer
  ^
  | STOP pointer

Class 3      +
  ^
  | START pointer
  ^
  | STOP pointer

Class 4      ,
  ^
  | START pointer
  ^
  | STOP pointer

Class 5      Symbol_or_Label
  ^
  | START pointer
  ^
  | STOP pointer

Class 7      lower_case_variable
  ^
  | START pointer
  ^
  | STOP pointer

Class 10     First GET_TOKEN 10E2
  ^
  | START pointer
  ^
  | STOP pointer

RESULT=10

Second GET_TOKEN 10E2
  ^
  | START pointer
  ^
  | STOP pointer

RESULT=2
```

**4-16 Assembler Subroutines**

## NOT\_DUPLICATE

Can be used in conjunction with UPDATE\_LABEL to prevent the assembler from marking a label as a duplicate. Normally, all labels are marked as a duplicate if they are used in the label field more than once. If the user wants the capability to redefine a label and assign it a different VALUE, this subroutine prevents the assembler from flagging the label as an error.

## PRINT\_LOCATION

Instructs the assembler to print the current value of the program counter on the source listing. Normally, this function is automatic when the subroutine GEN\_CODE is called, but if an instruction does not generate code, then this subroutine can be used.

## SAVE\_ERROR

An error message is displayed from the same list used for ERROR. The SAVE\_PTR pointer is used as the error message pointer in the assembler listing and it must be correctly positioned by the programmer.

Example:

```
MVI XX,LABEL                ;XX is an invalid operand
    ^
    | SAVE_PTR pointer
ERROR-IO ^
    |
    | Error message in the assembler
    | listing.
```

## SAVE\_WARNING

A warning message is displayed from the same list used for ERROR. The SAVE\_PTR pointer is used as the warning message pointer in the assembler listing and it must be correctly positioned by the programmer.

Example:

```
MVI XX,LABEL                ;XX is an invalid operand.
    ^
    | SAVE_PTR pointer
WARNING-IO ^
    |
    | Warning message in the
    | assembler listing.
```

## SCAN\_REAL

Converts real decimal numbers to binary real numbers. All assemblers currently have a REAL pseudo instruction that converts real decimal numbers to the IEEE standard for short or long real binary numbers. If this is not the encoding desired, SCAN\_REAL in the User Definable Assembler can be used to

parse real numbers and generate them in any binary pattern. Exponents can be up to 16 bits and mantissas can be up to 64 bits.

SCAN\_REAL is called in the same manner as other User Definable Assembler instructions and uses some of the temporary registers (TEMP38 through TEMP40). It expects the STOP pointer to be positioned at the beginning of a real decimal number (refer to the explanation of the REAL pseudo in the *Assembler/Linker Reference Manual* for real number syntax).

Example:

```

1.23E2
^
| STOP pointer
;Equals 123 decimal

```

Temporary registers 38 through 40 are used to pass information to the SCAN\_REAL routine and to obtain converted data.

```

MANTISSA_SIZE = TEMP38    ;Pass mantissa size to
                           ;SCAN_REAL.
EXPONENT       = TEMP38    ;Exponent passed from
                           ;SCAN_REAL.
MANTISSA_HI    = TEMP39    ;Upper 32 bits of mantissa
                           ;from SCAN_REAL.
MANTISSA_LO    = TEMP40    ;Lower 32 bits of mantissa
                           ;from SCAN_REAL.

```

Mantissa size (TEMP38) is initialized before the call to SCAN\_REAL to indicate the bit size of the mantissa field for rounding purposes (maximum 64). The SCAN\_REAL instruction can then be called to convert the decimal real number. If no syntax errors were found, then the results of the conversion will be in TEMP38 - TEMP40. If there is an error, a return 1 is executed and the stop pointer is not incremented. TEMP38 will hold the binary exponent, TEMP39 the upper 32 bits, and TEMP40 the lower 32 bits of the normalized mantissa. These results can be arranged and output in any manner. Example:

Assume that we will be converting a decimal number to a binary real number with a 50-bit mantissa and the STOP pointer positioned as follows.

```

1.23E2
^
| STOP pointer
;Decimal 123

```

The code would look something like:

#### 4-18 Assembler Subroutines

```

LOAD 50                                ;Set mantissa size.
STORE TEMP38

SCAN_REAL                               ;Convert decimal number.
GOTO NOT_REAL                           ;Return 1- real number expected.
                                        ;and not found.

--                                       ;Return 2- real number found
                                        ;and converted.

```

### Results:

```

TEMP38 = 00000006 ;Size of binary exponent.
TEMP39 = F6000000 ;Normalized high part of mantissa.
TEMP40 = 00000000 ;Low part of mantissa.

```

### Note




---

SCAN\_REAL will not parse minus signs in front of decimal numbers. Check for these before calling SCAN\_REAL.

---

## UPDATE\_LABEL

Allows the user to redefine the VALUE and TYPE of the label on the current source statement. The main purpose of this subroutine is to allow the user to assign attributes to symbols and still permit the label to be relocatable. The lower four bits of the TYPE must not be changed; however, the upper 28 bits can be used to assign attributes to the label. These attributes will be carried with the symbol and returned when the EXPRESSION or GET\_SYMBOL subroutines are used.

## WARNING

A warning message is displayed from the same list used for ERROR. The START pointer is used as the warning message pointer in the assembler listing and it must be correctly positioned by the programmer.

Example:

```

MVI XX,LABEL                            ;XX is an invalid operand.
  ^
  | START pointer
WARNING-IO  ^
            |
            |

```

---

## Notes





## Creating An Assembler

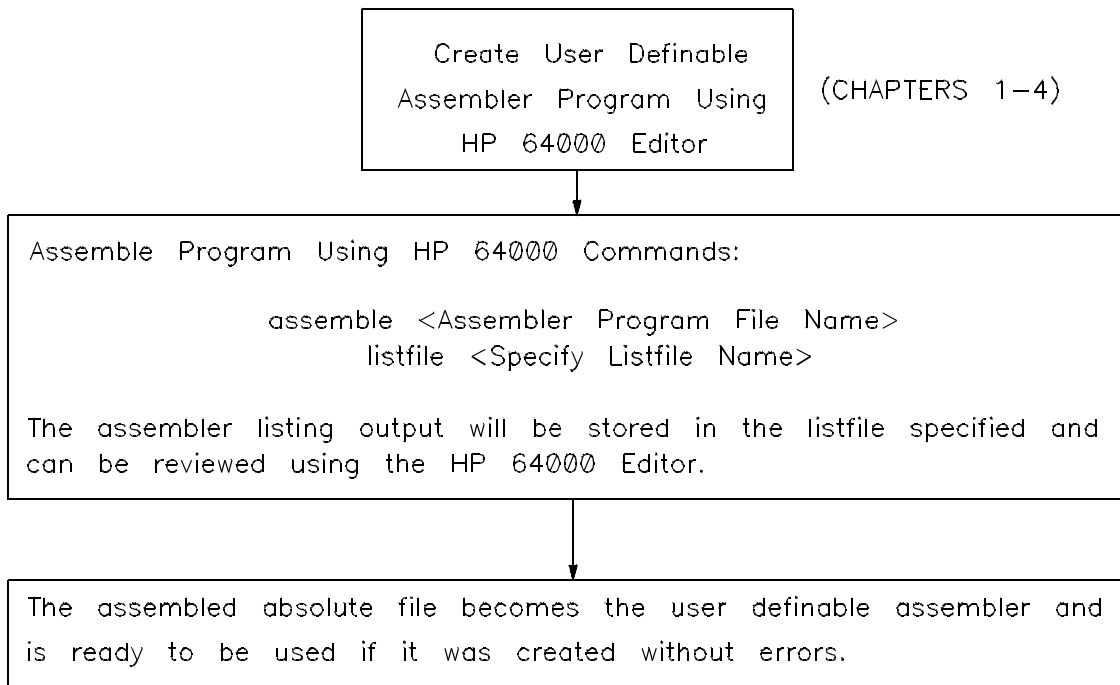
---

### Introduction

This chapter explains how to create the user definable assembler source program after the target processor has been completely defined. The assembler program is treated like any other source program, except the output of the assembly process is in absolute format, eliminating the need for a linking sequence. The program is stored in a Model 64000 absolute file to be used to assemble any user target program for the defined microprocessor. Figure 5-1 indicates the sequence of events that occur when creating a user definable source program.

If further explanation is needed, a summary of the building process using the 8080 processor starts after figure 5-1. In Appendix A, the complete assembler code is included. Note that the source line numbers (SN) in the summary examples match those in the complete code in Appendix A.

Also included in this chapter is an example of the TRACE pseudo instruction. This instruction enables the user to examine execution of the user definable assembler program after it has been assembled.



**Figure 5-1. Creating the Assembler**

---

## Summary Of The Assembler Source Code Building Process for 8080 Processor

### **Assembler Setup Commands**

In defining a processor, the first statement must be the ASSEMBLER setup command followed by the assembler directive in quotation marks. For example:

```
ASSEMBLER "8080"
```

### 5-2 Creating an Assembler

Following the assembler directive statement, the basic parameters of the user processor are defined with the setup commands. These parameters include such things as word size, address size, assembler list title, print field size, linker file identifier, registers, status words, and stack pointers. Some examples of setup commands that may be entered after the ASSEMBLER directive follow.

Example:

```
SN
1  ASSEMBLER "8080"           ;Defines the processor.
.
.
9   WORD_SIZE = 8             ;Defines the word size.
10  ADDRESS_BASE = 8         ;Specifies the program
11                                ;counter increment.
12  TITLE = "8080"           ;Title for the assembler list.
13  LOC_SIZE = 4             ;Four characters in the print field for the
                                ;location counter.
14  LINK_FILE L8080 : XX     ;Specifies linker file.
                                ;XX is the USERID (1 to 6 characters).
15  PC_16                     ;Only the lower 16 bits
                                ;of the program counter are used.
17  RELOC_FMT HIGH_LOW, SIZE=16 ;Relocate 16 bits.
18  RELOC_FMT LOW_HIGH, SIZE=16 ;Relocate and swap bytes.
19  RELOC_FMT LOW_BYTE, SIZE=8  ;Low byte, no error check.
20  RELOC_FMT HIGH_BYTE, SIZE=8 ;High byte, no error check.
21  RELOC_FMT LOW_CHECK, SIZE=8 ;Low byte, check for >256.
22  RELOC_FMT REL_8, SIZE=8     ;Plus minus 128.
23  RELOC_FMT PC_REL, SIZE=8    ;-126, +129
```

After the assembler setup commands have been established, the user must identify predefined registers, stack pointers, condition codes, etc., that are relevant to the specified processor. Using the assembler directive listing above as a base, the additional information about the specified microprocessor should be entered into the program as follows:

Example:

```
SN
1  ASSEMBLER "8080"           ;Defines the processor.
.
.
9   WORD_SIZE = 8             ;Defines the word size.
10  ADDRESS_BASE = 8         ;Specifies the program
11                                ;counter increment.
12  TITLE = "8080"           ;Title for the assembler list.
```

```

13  LOC_SIZE = 4                ;Four characters in the
                                ;print field for the location counter.
14  LINK_FILE L8080 : XX       ;Specifies linker file.
                                ;XX is the USERID (1 to 6 characters).
15  PC_16                       ;Only the lower 16 bits
                                ;of the program counter are used.
17  RELOC_FMT HIGH_LOW, SIZE=16 ;Relocate 16 bits.
.
.
.
25  CONSTANTS
26  HIGH_FLAG = TEMP1          ;Used as a flag if HIGH
                                ;keyword is found.
27  COUNT = TEMP2              ;Used as a temporary count.
28  MEM_CHECK = TEMP3          ;Used to check memory
                                ;reference on MOV instructions.
29  END

30
31  SYMBOLS = REGISTER          ;Defines the TYPE and
                                ;VALUE assigned to the
32  A=7                         ;symbols. REGISTER is
                                ;TYPE 6. Symbol C has
33  B=0                         ;a VALUE of 1.
34  C=1

.
.
.
40  END

SYMBOLS = XXX_XXX              ;Continue to add symbol
.                               ;tables, such as condition
.                               ;codes, where applicable
.                               ;for the processor.
.                               ;Terminate each table with
.                               ;END instruction.
END

```

## Defining and Parsing the Instruction Set (INSTR\_DEF & INSTR\_SET)

The instruction set must be divided into separate groups of instructions that are parsed in the same way by using the command INSTR\_DEF. After INSTR\_DEF, each instruction should be listed with its object code format. Next, the command INSTR\_SET implements the instruction group parsing rules defined for the user processor. After a group (or a single instruction if it is unique) is defined by INSTR\_DEF and INSTR\_SET, the section is terminated by assembler instruction END. Continuing with the same sample program, implement INSTR\_DEF and INSTR\_SET as follows.

### 5-4 Creating an Assembler

### Example:

```
SN
1  ASSEMBLER "8080"           ;Defines the processor.
.
.
9   WORD_SIZE = 8             ;Defines the word size.
10  ADDRESS_BASE = 8         ;Specifies the program
.                               ;counter increment.
.
.
17  RELOC_FMT HIGH_LOW, SIZE=16 ;Relocate 16 bits.
.
.
.
SYMBOLS = XXX_XXX           ;Continue to add symbol
.                               ;tables, such as condition
.                               ;codes, where applicable
.                               ;for the processor.
.                               ;Terminate each table with
.                               ;END instruction.
END

55  INSTR_DEF OPERAND=0
.
.
61  CMC = 3FH                ;list of no operand
.                               ;instructions
.
.
.
85  HLT = 76H
.
.
87  INSTR_SET
88
89  GEN_CODE ABS 8, OBJECT_CODE
90  DONE
91
92  END
.
.
.
.
```

Continue building INSTR\_DEF/INSTR\_SET tables until all instructions for the target processor are defined. Refer to Appendix A for complete user defined assembler code for 8080 processor.

## Tracing The User Defined Assembler Execution Sequence

The TRACE pseudo instruction allows the user to examine the execution of the user defined assembler program. With it, the user can obtain a printout of the contents in the program counter, accumulator, and VALUE and TYPE variables. TRACE 1 traces pass 1, TRACE 2 traces pass 2, TRACE 3 traces both passes, and TRACE 0 disables the TRACE pseudo. Figure 5-2 shows a sample output from an 8080 source program using the TRACE 2 pseudo instruction. Refer to Appendix A for the complete 8080 source program.

```
HEWLETT-PACKARD: 8080 ASSEMBLER

LOCATION OBJECT CODE LINE      SOURCE LINE
1  8080R2      TRACE 2
3 P 00000000 0066 EXP   A=00000000 V=00000007 T=0006 START=12 STOP=12
3 P 00000000 0067 IF    A=00000000 V=00000007 T=0006 START=12 STOP=12
3 P 00000000 006A LOAD   A=00000007 V=00000007 T=0006 START=12 STOP=12
3 P 00000000 006B LEFT   A=00000038 V=00000007 T=0006 START=12 STOP=12
3 P 00000000 006C OR    A=0000003E V=00000007 T=0006 START=12 STOP=12
3 P 00000000 006D ST     A=0000003E V=00000007 T=0006 START=12 STOP=12
3 P 00000000 006E COMA  A=0000003E V=00000007 T=0006 START=12 STOP=12
3 P 00000000 0070 CALL  A=0000003E V=00000007 T=0006 START=12 STOP=13
3 P 00000000 0085 ST_0  A=0000003E V=00000007 T=0006 START=12 STOP=13
3 P 00000000 0086 SYMB  A=0000003E V=00000007 T=0006 START=13 STOP=13
3 P 00000000 0087 GOTO  A=0000003E V=00000007 T=0006 START=13 STOP=13
3 P 00000000 008F LOAD  A=00002B6B V=00000007 T=0006 START=13 STOP=13
3 P 00000000 0090 ST     A=00002B6B V=00000007 T=0006 START=13 STOP=13
3 P 00000000 0070 RET   A=00002B6B V=00000007 T=0006 START=13 STOP=13
3 P 00000000 0071 EXP   A=00002B6B V=00000020 T=0000 START=13 STOP=16
3 P 00000000 0072 CODE  A=00002B6B V=00000020 T=0000 START=13 STOP=16
3 P 00000001 0073 IF    A=00002B6B V=00000020 T=0000 START=13 STOP=16
3 P 00000001 0076 CALL  A=00002B6B V=00000020 T=0000 START=13 STOP=16
3 P 00000001 0092 COMA  A=00002B6B V=00000020 T=0000 START=13 STOP=16
3 P 00000001 0076 RET   A=00002B6B V=00000020 T=0000 START=13 STOP=16
3 P 00000001 0077 IF    A=00002B6B V=00000020 T=0000 START=13 STOP=16
3 P 00000001 007A CODE  A=00002B6B V=00000020 T=0000 START=13 STOP=16
3 P 00000002 007B GOTO  A=00002B6B V=00000020 T=0000 START=13 STOP=16
3 P 00000002 0015 EOL   A=00002B6B V=00000020 T=0000 START=13 STOP=16
3 P 00000002 0016 DONE  A=00002B6B V=00000020 T=0000 START=13 STOP=16
0000 3E20          3      MVI   A,20H
4      TRACE 0
ERRORS= 0
```

Figure 5-2 Example of TRACE 2 Output

## 5-6 Creating an Assembler

KEY:      1st column is source code line number (here 3)  
          2nd column is program counter in use (here P-PROG)  
          3rd column is contents of user program counter  
          4th column is assembler instruction location  
          5th column is assembler instruction abbreviation  
          6th column is contents of accumulator  
          7th column is VALUE  
          8th column is TYPE  
          9th column is location of START pointer  
         10th column is location of STOP pointer

**Figure 5-2 Example of TRACE 2 Output (cont.)**



---

## Notes





## Linker General Information

---

### Introduction

A linker combines the relocatable object files generated by the assembler into one file, producing an absolute image that will load and execute within a specified area of physical memory.

### Note



---

If the user already has a 64000 Assembler/Linker for the target processor, there is no need to define a linker program. The existing 64000 linker absolute file can be used unless additional relocatable formats are added to the assembler. It will be located in the 64000 directory under the processor name; e.g., I8085\_Z80 : HP.

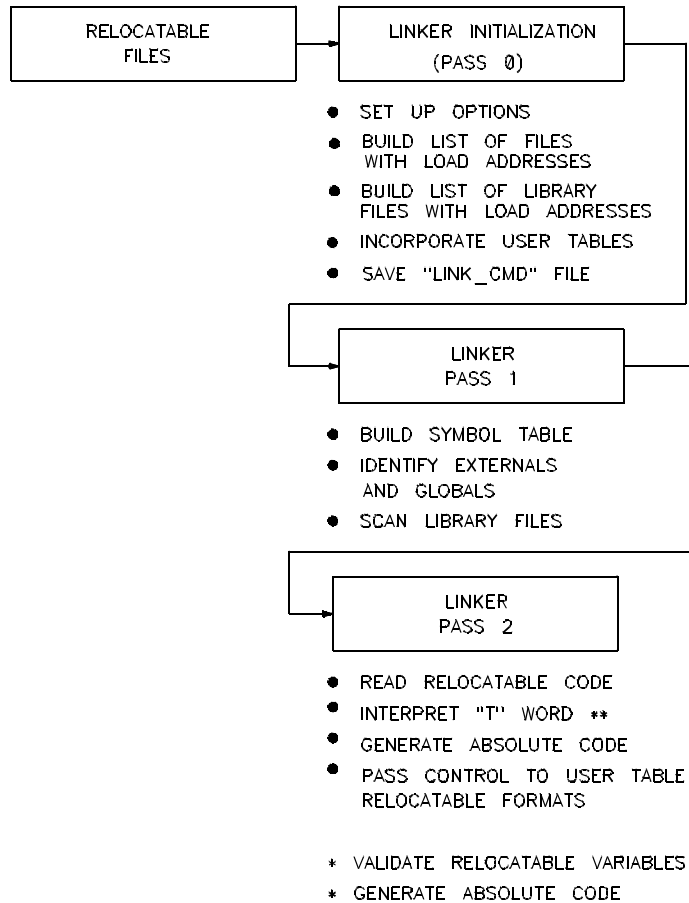
---

---

### Linker Operation

As mentioned in Chapter 1, the user definable linker has two modules, the basic linker module and the user definable linker module. The functions performed by these modules are shown in figure 6-1. It is obvious most of the linker functions are performed by the basic linker module that is part of the operating system. The user definable linker module tailors the basic linker module for the target processor.

Certain operations such as performing range checks on the value of an external variable or merging this value with the opcode part of the instruction can only be performed by the user definable linker module. The value of an external variable is not available to the assembler.



- BASIC LINKER MODULE
- \* USER DEFINABLE
- \*\* REFER TO DOUBLE RECORD FILE FORMAT (DBL) IN APPENDIX D

**Figure 6-1. Linker Module Functions**

**6-2 Linker General Information**

## Linker Programming Rules

---

### Linker Structure

The linker structure is similar to the assembler except there are only three sections to be defined by the user. First, the user processor structure is defined by word size, minimum addressable unit (byte or word), number of bits necessary to specify an address, etc. This is accomplished with the linker setup commands. Next, entry points for relocatable routines that will handle the relocatable formats listed in the assembler are defined. Finally, the routines to handle the relocatable code created by the user defined assembler are defined with linker instructions and predefined symbols.

The functional block diagram in figure 7-1 illustrates the linker building process. Each block corresponds to a paragraph title. Sample linker code for the 8080 processor will be used in the explanations. Appendix B contains a complete listing of user defined linker code for the 8080 processor. Note that the source line numbers (SN) in the examples match those in the complete listing in Appendix B.

### Caution

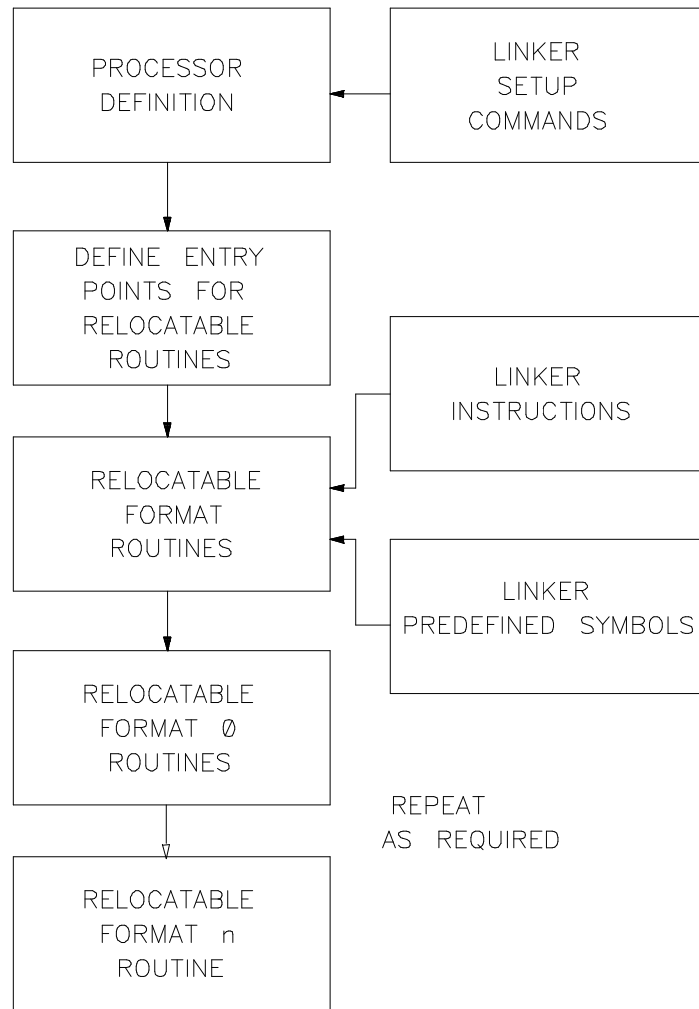


---

The order in which the linker table is constructed is critical to linker operation. Parts of the table can be omitted and no errors will be flagged. Refer to Appendix B for a complete table.

---

Recall that in the assembler, one of the setup commands was ASSEMBLER "8080", which defined the user processor in the example shown in Chapter 5. There is no directive in the linker structure, only a general "LINK" command that identifies the file as a linker. A virtual processor is used and the setup command in the assembler, LINK\_FILE L8080 : USERID, specifies the processor.



**Figure 7-1. Linker Building Process**

**7-2 Linker Programming Rules**

---

## Linker Setup Commands

Use the following setup commands to define the processor structure.

ALIGN	aligns PROG, DATA, and COMMON for each relocatable module by incrementing the load address until: load address AND ALIGN = 0
BASE	smallest addressable unit in bits.
DIGITS0	number of digits to be displayed in pass 0 (initialization).
DIGITS2	number of digits to be displayed in pass 2 (load map)
DBLADR	if set true, treats the program counter as a 32-bit quantity. If set false, all arithmetic operations will only affect the lower 16 bits of the program counter.
HISHIFT	number of bits the high order word has to be shifted to perform the internal/external address conversion.
IDOFFSET	system global describing the number of VALUE words in a symbol (see Appendix D-DBL record). If IDOFFSET = 2 1 word of symbol value DBLADR = false If IDOFFSET = 3 2 words of symbol value DBLADR - true
IND	allows the linker to automatically build indirect links in base page for processors that allow indirect memory addressing modes.

MAXL MAXH	maximum address range allowed during the initialization phase of the linker (pass 0). MAXL contains the least significant 16 bits. MAXH contains the most significant 16 bits.
MULTISPACE	allows programmer to use high order address bits to describe multiple spaces. Note, the user must mask load addresses and symbol addresses internal to the linker.
SWAP	exchanges positions of upper and lower bytes when absolute code is generated.
WIDTH	word size in bits.

## Processor Definition

Using the 8080 processor as an example, it is defined as follows:  
 Word size = 8 bits  
 Minimum addressable unit = 8 bits (byte)  
 Bits to define an address = 16

In the next section is sample code defining the 8080 processor. Become familiar with the linker setup commands before examining the sample code.

### Sample Code Defining 8080 Processor

The sequence of linker setup commands that must be used is shown in the following sample source code. The sequence cannot be altered and the number of definition words must total 32 (20H). The pseudo instruction HEX is used to store information in the hexadecimal format (refer to *Assembler/Linker Reference Manual* for details). Note the next statement after the definition words must be the length of the table: DEF LAST-\$. LAST is the last instruction in the table. Refer to Appendix B, where the complete user defined linker source code for the 8080 processor is included.

```

SN
1  "LINK"
.
.
5  HEX 10                                ;Number of valid constants.
                                        ;In lines 6 through 21 there
                                        ;are 16 constants (10H).
6  HEX 2  IDOFFSET                        ;1 word of symbol value, DBLADR is false.
7  HEX 8  WIDTH                          ;8-bit words.
8  HEX 8  BASE                            ;Byte addressable.
9  HEX 0  ALIGN                           ;ALIGN is 0. 8080 does not
                                        ;need to be word aligned.
10 HEX 5  DIGITS0                         ;Number of digits to display
                                        ;in pass 0. Need 5H digits
                                        ;to put 16-bit address.
11 HEX 4  DIGITS2                         ;Number of digits to display
                                        ;in pass 2. Need 4H digits
                                        ;to output 16-bit address.
12 HEX 0  DBLADR                          ;DBLADR is false.
13 HEX 0  SWAP                            ;No byte swapping.
14 HEX 0  IND                             ;No memory indirect addressing.
15 HEX 0  MULTISPACE                      ;True if multiple address spaces.
16 HEX FFFF MAXL                          ;Maximum legal address in
17 HEX 0  MAXH                            ;pass 0 is 0FFFFH.
18 HEX 0  UNDEF                           ;Included to keep
19 HEX 0  UNDEF                           ;word count
20 HEX 0  UNDEF                           ;correct.
21 HEX 0  HISHIFT                         ;Upper word need not be shifted for internal/
                                        ;external address conversion.
22 HEX 0  UNDEF                           ;Included to complete
23 HEX 0  UNDEF                           ;word count
24 HEX 0  UNDEF                           ;of 32.
25 HEX 0  UNDEF                           .
26 HEX 0  UNDEF                           .
27 HEX 0  UNDEF                           .
28 HEX 0  UNDEF                           .
29 HEX 0  UNDEF                           .
30 HEX 0  UNDEF                           .
31 HEX 0  UNDEF                           .
32 HEX 0  UNDEF                           .
33 HEX 0  UNDEF                           .
34 HEX 0  UNDEF                           .
35 HEX 0  UNDEF                           .
36 HEX 0  UNDEF                           .
37
.
.
41 DEF    LAST-$                          ;Word length location must be
                                        ;at 20H (See Appendix B).

```

## Linker Programming Rules 7-5

---

## Define Entry Points For Relocatable Routines

Back in Chapter 2, "Defining The Processor", relocatable formats were defined with the RELOC\_FMT setup command. These formats must now be handled with the linker instructions and predefined symbols. The first step is to define the entry points for routines that will handle each relocatable format listed in the assembler. It is essential that the same sequence used in the assembler be followed. The linker instruction DEF is used to define the entry points for the routines. The relocatable formats in Chapter 2 are repeated here with their DEF instructions. Linker instructions and predefined symbols are listed after this section. An explanation of the relocatable routines then follows.

```
RELOC_FMT HIGH_LOW, SIZE = 16      DEF FMT0
RELOC_FMT LOW_HIGH, SIZE = 16     DEF FMT1
RELOC_FMT LOW_BYTE, SIZE = 8      DEF FMT2
RELOC_FMT HIGH_BYTE, SIZE = 8     DEF FMT3
RELOC_FMT LOW_CHECK, SIZE = 8     DEF FMT4
RELOC_FMT REL_8, SIZE = 8         DEF FMT5
RELOC_FMT PC_REL, SIZE = 8       DEF FMT6
```

Formats FMT0 and FMT1 will be explained for illustration. The source line numbers (SN) match those in the complete code in Appendix B.

```
SN
42 DEF FMT0 ;Two-byte address, HI,LO.
43 DEF FMT1 ;Two-byte address, LO,HI.
```

---

## Linker Instructions

Use these linker instructions to write the relocatable format routines.

**ADD op1,op2,op3**      adds the contents of operand 3 to the contents of operand 2 and returns the result in operand 1.  
op1 <-- op2 + op3

**AND op1,op2,op3**      logically ANDs the contents of operand 3 with the contents of operand 2 and returns the result in operand 1.  
op1 <-- op2 AND op3

### 7-6 Linker Programming Rules



BLDLINK	creates indirect addressing links in a predefined area of memory if IND has been set. Finds predefined symbol LLA and loads ADR into LLA.
CALL label	transfers program control to subroutine label. Only one level of subroutines is allowed.
DEF expression	pseudo instruction that allows the definition of expressions typically used with immediate op1 instructions.
DONE	returns control to the basic linker module and generates absolute code.
ERROR "..." WARNING "..."	creates the error or warning message as defined by the immediate ASCII string.
GOTO label	transfers program control to the instruction following the label.
IMMEDIATE op1	loads the value of the constant specified in the next program line into operand 1. op1 <-- constant
IOR op1,op2,op3	performs an inclusive OR function on the contents of operand 2 and operand 3 and returns the result in operand 1. op1 <-- op2 IOR op3
LOADBYTES n	loads the n least significant bytes of LOADWRD into the output buffer.
LOADBITS n	loads the n least significant bits of LOADWRD into the output buffer.
MOVE op1,op2	moves the contents of operand 2 into operand 1. op1 <-- op2

## Linker Programming Rules 7-7

ONECMP op1,op2	computes the one's complement of operand 2 and returns the result into operand 1. op1 <-- op2
RETURN n	returns to location n past CALL.
SEQ op1,op2	skips the next instruction if operand 1 is equal to operand 2.
SEQZ op1	skips the next instruction if operand 1 is equal to zero.
SGE op1,op2	skips the next instruction if operand 1 is greater than or equal to operand 2.
SHIFTL n,op1,op2	shifts the contents of operand 2, n bits to the left and returns the result in operand 1. n = 1 to 16.
SHIFTR n,op1,op2	shifts the contents of operand 2, n bits to the right and returns the result in operand 1. n = 1 to 16.
SKELETON	loads the skeleton of the object code into LOADWRD.
SNEZ op1	skips the next instruction if operand 1 is not equal to zero.
SWAPBYTES op1,op2	interchanges the upper byte with the lower byte in the least significant 16 bits of operand 2 and returns the result in the least significant 16 bits of operand 1.
SWAPWORDS op1,op2	interchanges the upper 16 bits with the lower 16 bits of operand 2 and returns the result in operand 1.

## 7-8 Linker Programming Rules

TRACE	prints the values of all the linker variables and registers plus the location code of the TRACE instruction. Helps debug linker code. TRACE must be inserted in the linker source code where required and then removed after the debugging phase is completed.
TWOCMP op1,op2	computes the two's complement of operand 2 and returns the result in operand 1. op1 <-- op2 + 1
XOR op1,op2,op3	performs an exclusive OR function on the contents of operand 2 and operand 3 and returns the result in operand 1. op1 <-- op2 XOR op3

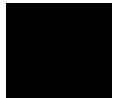
**Note**




---

Operands op1,op2, and op3 must be one of the following predefined symbols.

---




---

**Predefined Symbols**

Use these predefined symbols to write the relocatable format routines.

ADR	absolute address of variable to be tested will be contained in ADR.
LLA	links load address. Used in conjunction with BLDLINK and IND.
LOADADR	contains the value of the program counter for the processor.

LOADWRD	machine code word output register. The linker will only generate absolute code with the contents of LOADWRD.
T0 through T3	temporary registers 0 through 3.

---

## Relocatable Format Routines

The entry points for the relocatable routines have been defined with the DEF linker instruction. Now the routines must be written using the linker instructions and predefined symbols to convert the relocatable code to absolute code. Routines FMT0 and FMT1 will be explained for illustration. The source line numbers (SN) match those in the complete code in Appendix B.

```

SN
50 FMT0 MOVE LOADWRD,ADR      ;Move the contents of ADR to LOADWRD.
51     LOADBYTES 2           ;Output two bytes of code
                                ;that is in LOADWRD.
52     DONE                  ;End of routine.
53 FMT1 SWAPBYTES LOADWRD,ADR ;Take absolute address in
                                ;ADR and store in LO HI format in LOADWRD.
54     LOADBYTES 2           ;Output two bytes of code
                                ;that is in LOADWRD.
55     DONE                  ;End of routine.

```

Appendix D lists the actual relocatable and absolute record file formats by word.

## Creating The Linker

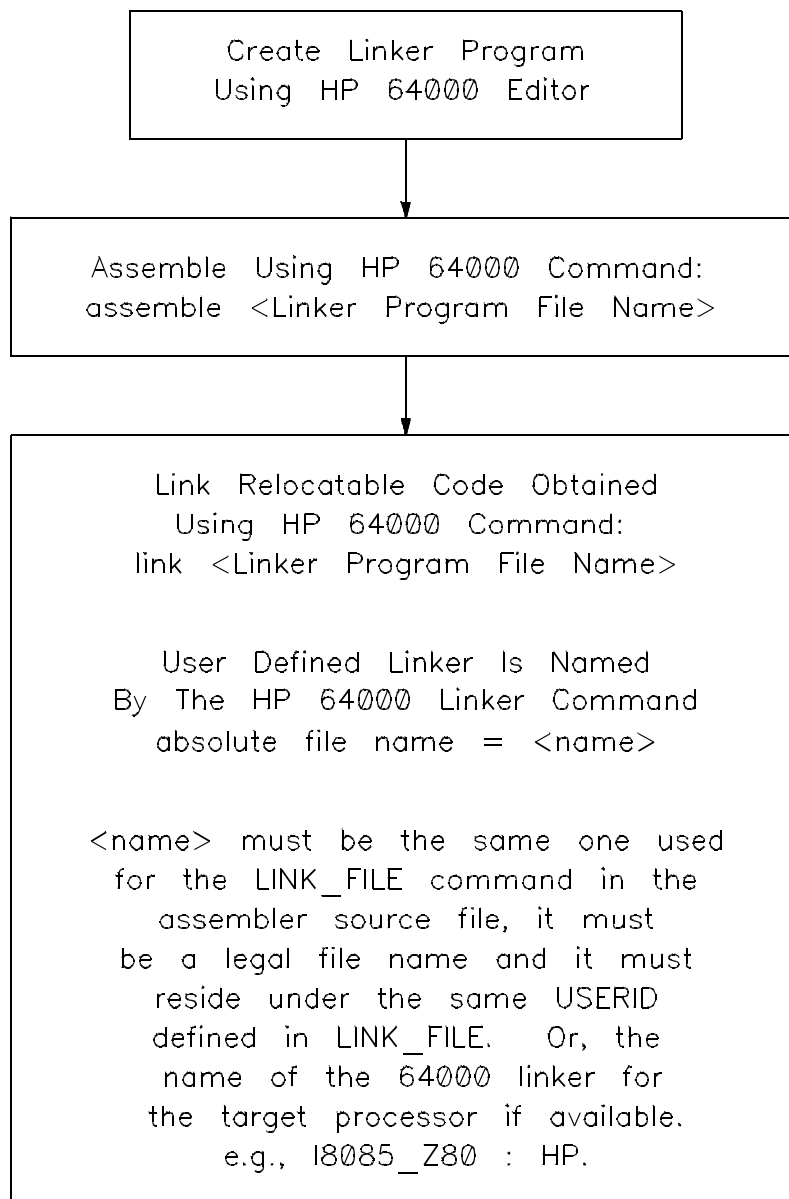
---

### Introduction

This section explains how to create the user definable linker program after the target processor has been defined. The program will then be stored in a Model 64000 absolute file for future use with the target processor. The program is generated by using the editor function of the Model 64000, following the structure defined in the previous sections. The program file constructed using the editor can now be assembled and linked into an absolute file just as any other source file, except for the use of the virtual processor "LINK". The user defined linker, now in the absolute file, will link the relocatable object code files for the target processor. Figure 8-1 illustrates the sequence of events that should be accomplished by the user.

Also included in this chapter is an example of the TRACE pseudo instruction. This instruction enables the user to examine execution of the user defined linker program.





**Figure 8-1. Creating the Linker**

**8-2 Creating the Linker**

## Tracing The User Defined Linker Execution Sequence

The TRACE instruction allows examination of the user defined linker code during execution. The instruction should not be inserted between IMMEDIATE and DEF or just after skip instructions. TRACE is used in the following example.

Example:

```

FILE: LTRACE: I8080      HEWLETT-PACKARD: USER DEFINABLE LINKER
                        OBJECT
LOCATION   CODE          LINE    SOURCE LINE
-----
0028     OC85          50 FMT0  MOVE LOADWRD,ADR      ;LOADWRD=LOADADR
0029     0056          51      LOADBYTES 2          ;LOAD 2 BYTES AND LOADBYTES,,
002A     0018          52      DONE
002B     0004          53 FMT1  TRACE
002C     OC88          54      SWAPBYTES LOADWRD,ADR ;LOADWRD=SWAPBYTES(LOADADR)
002D     0004          55      TRACE
002E     0056          56      LOADBYTES 2          ;LOAD 2 BYTES AND LOADBYTES,,
002F     0004          57      TRACE
0030     0018          58      DONE
0031     OC85          59 FMT2  MOVE LOADWRD,ADR      ;LOADWRD=LOADADR
0032     0036          60      LOADBYTES 1          ;LOAD 1 BYTE AND LOADBYTES,,
0033     0018          61      DONE

```

The output will contain the following information.

```

HP 64000 LINKER 2.0                      MON, 9 MAR 19  , 10:30
-----
FILE/PROG  NAME          PROGRAM  DATA    COMMON  ABSOLUTE  DATE          TIME
-----
TRACE:I8080          1000
TRACE AT 002BH LOADADR=1000H LOADWRD=9726H ADR=1002H T0=0000H T1=000H T2=000H T3=000H LLA=000H
TRACE AT 002DH LOADADR=1000H LOADWRD=0210H ADR=1002H T0=0000H T1=000H T2=000H T3=000H LLA=000H
TRACE AT 002FH LOADADR=1002H LOADWRD=0210H ADR=1002H T0=0000H T1=000H T2=000H T3=000H LLA=000H
next address          1002

XFER address= 0000      Defined by  DEFAULT
absolute & link_com file name=TRACE:I8080
Total# of bytes loaded 0002

```

---

## Notes





## Uploading To The Mainframe

---

### Introduction

The user defined assembler and linker tables you have created will be used by either the Model 64000 development station or the mainframe. The following instructions will explain how to upload the tables to the mainframe. Following these steps, your custom assembler will be ready for use in the HP-UX environment.

---

### Uploading Assembler Tables

After you have created your user defined assembler table source and assembled it, the resulting table is in absolute format with HP userid and a name beginning with a capital A. For example:

A directive in the UDA source ASSEMBLER "68000" would create the assembler table A68000:HP:absolute.

To upload the assembler table to the mainframe, use the file transfer utility of the Hosted Development System in either the RS232 or High-Speed Link mode to the /usr/hp64000/tables directory. For example, using RS232:

```
transfer -fab A68000:HP:absolute  
/usr/hp64000/tables/a68000
```

Using high-speed link:

```
transfer -fha A68000:HP:absolute  
/usr/hp64000/tables/a68000
```

### Uploading Linker Tables

After you have created your user defined linker table source and assembled it, you should create an absolute file using the linker with a name starting with "L" in the HP userid. The name must be

the same one used for the LINK\_FILE command in the assembler source file. For example:

```
L68000:HP:absolute
```

To upload the linker table to the mainframe, use the file transfer utility of the Hosted Development System in either the RS232 or High-Speed Link mode to the /usr/hp64000/tables directory. For example, using RS232:

```
transfer -fab L68000:HP:absolute  
/usr/hp64000/tables/l68000
```

Using high-speed link:

```
transfer -fha L68000:HP:absolute  
/usr/hp64000/tables/l68000
```

## Note



---

For more details on uploading, refer to "Using The File Transfer Utility" chapter in the *Users Guide - Hosted Development System*.

---

## User Defined Assembler Code for 8080 Processor

---

Assembler: A8080:HP                    64000 User Definable Assembler Utility

```

1 ASSEMBLER "8080"
2
3 ;*****;
4 ;
5 ;   64840-10002 - 8080 Assembler   ;
6 ;
7 ;*****;
8
0008 9 WORD_SIZE = 8           ;8 bit processor
0008 10 ADDRESS_BASE = 8      ;byte addressing
11
12 TITLE = "8080 Assembler"
0004 13 LOC_SIZE = 4
14 LINK_FILE 18085_Z80 : HP
15 PC_16
16
0000 17 RELOC_FMT HIGH_LOW, SIZE = 16   ;Relocate 16 bits
0001 18 RELOC_FMT LOW_HIGH, SIZE = 16   ;Relocate and swap bytes
0002 19 RELOC_FMT LOW_BYTE, SIZE = 8    ;low byte, no error check
0003 20 RELOC_FMT HIGH_BYTE, SIZE = 8   ;high byte, no error check
0004 21 RELOC_FMT LOW_CHECK, SIZE = 8    ;low byte, check for > 256
0005 22 RELOC_FMT REL_8, SIZE = 8       ;plus minus 128
0006 23 RELOC_FMT PC_REL, SIZE = 8      ;-126, +129
24
25 CONSTANTS
001C 26 HIGH_FLAG = TEMP1   ;Use as flag if HIGH keyword found
001E 27 COUNT = TEMP2     ;Use as a temporary count
0020 28 MEM_CHECK = TEMP3   ;Used to check memory reference on
                          ;MOV instructions
29 END
30
0006 31 SYMBOLS = REGISTER
0007 32   A = 7
0000 33   B = 0
0001 34   C = 1
0002 35   D = 2
0003 36   E = 3

```

Assembler: A8080:HP            64000 User Definable Assembler Utility

```
0004 37 H = 4
0005 38 L = 5
0006 39 M = 6
      40 END
      41
0007 42 SYMBOLS = STATUS
0006 43 PSW = 6
      44 END
      45
0008 46 SYMBOLS = STACK
0006 47 SP = 6
      48 END
      49
0009 50 SYMBOLS = ADDR_OPER
0001 51 HIGH = 1
0000 52 LOW = 0
      53 END

0000 55 INSTR_DEF    OPERAND = 0
      56
      57 ;*****;
      58 ; No operand instructions ;
      59 ;*****;
      60

003F 61 CMC = 03FH
0037 62 STC = 37H
002F 63 CMA = 2FH
0000 64 NOP = 0
0027 65 DAA = 27H
0007 66 RLC = 7
000F 67 RRC = 0FH
0017 68 RAL = 17H
001F 69 RAR = 1FH
00EB 70 XCHG = 0EBH
00E3 71 XTHL = 0E3H
00F9 72 SPHL = 0F9H
00E9 73 PCHL = 0E9H
00C9 74 RET = 0C9H
00D8 75 RC = 0D8H
00D0 76 RNC = 0D0H
00C8 77 RZ = 0C8H
00C0 78 RNZ = 0C0H
00F8 79 RM = 0F8H
00F0 80 RP = 0F0H
00E8 81 RPE = 0E8H
00E0 82 RPO = 0E0H
00FB 83 EI = 0FBH
00F3 84 DI = 0F3H
0076 85 HLT = 76H
      86
```

## A-2 User Defined Assembler Code for 8080 Processor

Assembler: A8080:HP            64000 User Definable Assembler Utility

```

      87 INSTR_SET
      88
0001   C014 89   GEN_CODE ABS 8, OBJECT_CODE
0002   000E 90   DONE
      91
      92 END
      93
      94
      95 INSTR_DEF
      96
      97 ;*****;
      98 ;   restart instruction   ;
      99 ;*****;
100
00C7 101   RST = 0C7H
102
103 INSTR_SET
0004   0000 104  EXPRESSION
0005   2D0C 105  IF TYPE <> 0 THEN SAVE_ERROR IO_ERR
0008   2D0A 106  IF VALUE >7 THEN SAVE_ERROR IO_ERR
000B   050A 107  LOAD VALUE
000C   0183 108  SHIFT_LEFT 3
000D   4012 109  GOTO GEN_PRINT
110
111 END

113 INSTR_DEF
114
115 ;*****;
116 : operand: reg 0-7           ;
117 ;   object code, xxRRRxxx   ;
118 ;*****;
119
0004 120   INR = 4
0005 121   DCR = 5
122
123 INSTR_SET
124
000F   801B 125  CALL GET_REGISTER
0010   050A 126  LOAD VALUE
0011   0183 127  SHIFT_LEFT 3
128 GEN_PRINT
0012   1514 129  OR   OBJECT_CODE
0013   1914 130  STORE OBJECT_CODE
0014   C014 131  GEN_CODE ABS 8, OBJECT_CODE
132 CHECK_END
0015   0014 133  CHECK_EOL
0016   000E 134  DONE
135 EOL_ENTRY
0017   0581 136  LOAD STOP
```

**User Defined Assembler Code for 8080 Processor A-3**

Assembler: A8080:HP            64000 User Definable Assembler Utility

```
0018     1980 137     STORE START
0019     0061 138     ERROR EE_ERR
001A     000E 139     DONE
         140
         141     GET_REGISTER
001B     0000 142     EXPRESSION
001C     2D0C 143     IF TYPE <> REGISTER THEN SAVE_ERROR IO_ERR
001F     01C1 144     RETURN
         145     END
         146
         147
         148     INSTR_DEF
         149
         150 ;*****;
         151 ;     operand: reg 0-7
         152 ;     object_code xxxxxRRR     ;
         153 ;*****;
         154
0080     155     ADD = 80H
0088     156     ADC = 88H
0090     157     SUB = 90H
0098     158     SBB = 98H
00A0     159     ANA = 0A0H
00AB     160     XRA = 0A8H
00B0     161     ORA = 0B0H
00B8     162     CMP = 0B8H
         163
         164     INSTR_SET
         165
0021     801B 166     CALL GET_REGISTER
0022     050A 167     LOAD VALUE
0023     4012 168     GOTO GEN_PRINT
         169
         170     END
        
         172     INSTR_DEF
         173
         174 ;*****;
         175 ;     operand: rp(b or d)
         176 ;     object code, xxRRxxxx     ;
         177 ;*****;
         178
0002     179     STAX = 2
000A     180     LDAX = 0AH
         181
         182     INSTR_SET
         183
0025     801B 184     CALL GET_REGISTER
0026     050A 185     LOAD VALUE
0027     1002 186     AND 2
```

#### A-4 User Defined Assembler Code for 8080 Processor

Assembler: A8080:HP            64000 User Definable Assembler Utility

```
0028     2D16 187     IF ACCUMULATOR <> VALUE SAVE_ERROR IO_ERR
002B     0183 188     SHIFT_LEFT 3
002C     4012 189     GOTO GEN_PRINT
         190     END
         191
         192
         193     INSTR_DEF
         194
         195     ;*****;
         196     ;     operand: rp b,d,h,sp             ;
         197     ;         xxRRxxxx                 +
         198     ;*****;
         199

         0009 200     DAD = 9
         0003 201     INX = 3
         000B 202     DCX = 0BH
                 203
                 204     INSTR_SET
                 205
002E     0000 206     EXPRESSION
002F     2D0C 207     IF TYPE = STACK THEN GOTO FOUND_SP
                 208     RP_ENTRY
0032     2D0C 209     IF TYPE <> REGISTER THEN GOTO SAVE_IO_ERROR
0035     2D0A 210     IF VALUE > 4 THEN GOTO SAVE_IO_ERROR
                 211     FOUND_SP
0038     050A 212     LOAD VALUE
0039     1006 213     AND 6
003A     2D16 214     IF ACCUMULATOR <> VALUE THEN GOTO SAVE_IO_ERROR
003D     0183 215     SHIFT_LEFT 3
003E     4012 216     GOTO GEN_PRINT
                 217
                 218     SAVE_IO_ERROR
003F     008A 219     SAVE_ERROR IO_ERR
0040     4012 220     GOTO GEN_PRINT
                 221
                 222     END

                 224     INSTR_DEF
                 225
                 226     ;*****;
                 227     ;     operand: rp b,d,h,psw             ;
                 228     ;         xxRRxxxx                 +
                 229     ;*****;
                 230
00C5     231     PUSH = 0C5H
00C1     232     POP = 0C1H
                 233
                 234     INSTR_SET
```

### User Defined Assembler Code for 8080 Processor A-5

```

                235
0042      0000 236      EXPRESSION
0043      2D0C 237      IF TYPE = STATUS THEN GOTO FOUND_SP
0046      4032 238      GOTO RE_ENTRY
                239
                240      END
                241
                242
                243      INSTR_DEF
                244
                245      ;*****;
                246      ; operand: rp b,d,h,sp , low,high ;
                247      ;   xxRRxxxx ;
                248      ;*****;
                249
0001      250      LXI = 1
                251
                252      INSTR_SET
                253
0048      0000 254      EXPRESSION
0049      2D0C 255      IF TYPE = STACK THEN GOTO LXI_SP
004C      2D0C 256      IF TYPE <> REGISTER THEN SAVE_ERROR IO_ERR
004F      2D0A 257      IF TYPE > 4 THEN SAVE_ERROR IO_ERR
                258      LXI_SP
0052      050A 259      LOAD VALUE
0053      0183 260      SHIFT_LEFT 3
0054      1514 261      OR OBJECT_CODE
0055      1914 262      STORE OBJECT_CODE
0056      0007 263      CHECK_COMMA
0057      405F 264      GOTO INVALID_DELIM
0058      0000 265      EXPRESSION
0059      2D0C 266      IF TYPE > 5 THEN SAVE_ERROR IO_ERR
005C      C014 267      GEN_CODE ABS 8, OBJECT_CODE
005D      E1E1 268      GEN_CODE LOW_HIGH VALUE
005E      4015 269      GOTO CHECK_END
                270
                271      INVALID_DELIM
005F      0581 272      LOAD STOP
0060      1980 273      STORE START
0061      004A 274      ERROR IO_ERR
0062      C014 275      GEN_CODE ABS 8, OBJECT_CODE
0063      E1E1 276      GEN_CODE LOW_HIGH VALUE
0064      000E 277      DONE
                278
                279      END

                281      INSTR_DEF

```



Assembler: A8080:HP                    64000 User Definable Assembler Utility

```
282 ;*****
283 ; operand: reg (0-7) , low or high ;
284 ;   xxRRRxxx immediate byte   ;
285 ;*****;
286
0006 287   MVI = 6
      288
      289   INSTR_SET
      290
0066 0000 291   EXPRESSION
0067 2D0C 292   IF TYPE <> REGISTER THEN SAVE_ERROR IO_ERR
006A 050A 293   LOAD VALUE
006B 0183 294   SHIFT_LEFT 3
006C 1514 295   OR OBJECT_CODE
006D 1914 296   STORE OBJECT_CODE
006E 0007 297   CHECK_COMMA
006F 405F 298   GOTO INVALID_DELIM
      299   MVI_ENTRY

0070 8085 300   CALL CHECK_HIGH_LOW
0071 0000 301   EXPRESSION
0072 C014 302   GEN_CODE ABS 8, OBJECT_CODE
0073 2D0C 303   IF TYPE > 5 THEN SAVE_ERROR IO_ERR
0076 8092 304   CALL CHECK_OLD_H
0077 2D1C 305   IF HIGH_FLAG = 1 THEN GOTO GEN_HIGH
007A E0E2 306   GEN_CODE LOW_BYTE VALUE
007B 4015 307   GOTO CHECK_END
      308
      309   GEN_HIGH
007C 2D0C 310   IF TYPE = 0 THEN GOTO GEN_HIGH_ABS
007F E0E3 311   GEN_CODE HIGH_BYTE VALUE
0080 4015 312   GOTO CHECK_END
      313
      314   GEN_HIGH_ABS
0081 050A 315   LOAD VALUE
0082 0148 316   SHIFT_RIGHT 8
0083 C016 317   GEN_CODE ABS 8, ACCUMULATOR
0084 4015 318   GOTO CHECK_END
      319
      320   CHECK_HIGH_LOW
0085 1D1C 321   STORE_0 HIGH_FLAG
0086 0001 322   GET_SYMBOL
0087 408F 323   GOTO NOT_OPER
0088 408F 324   GOTO NOT_OPER
0089 2D0C 325   IF TYPE<> ADDR_OPER THEN GOTO NOT_OPER
008C 050A 326   LOAD VALUE
008D 191C 327   STORE HIGH_FLAG
008E 0005 328   GET_TOKEN
      329
```

**User Defined Assembler Code for 8080 Processor A-7**

```

                                330 NOT_OPER
008F    0580 331    LOAD START
0090    1981 332    STORE STOP
0091    01C1 333    RETURN
                                334
                                335 CHECK_OLD_H
0092    0007 336    CHECK_COMMA
0093    01C1 337    RETURN
0094    000A 338    GET_STOP_CHAR
0095    409B 339    GOTO H_ERROR
0096    2D0E 340    IF CHARACTER <> H THEN GOTO H-ERROR

0099    211C 341    STORE_1 HIGH_FLAG
009A    01C1 342    RETURN
                                343 H_ERROF
009B    004A 344    ERROR IO_ERROR
009C    01C1 345    RETURN
                                346
                                347 END
                                348
                                349
                                350 INSTR_DEF
                                351
                                352 ;*****;
00C6    00C6 357    ADI = 0C6H
00CE    00CE 358    ACI = 0CEH
00D6    00D6 359    SUI = 0D6H
00DE    00DE 360    SBI = 0DEH
00E6    00E6 361    ANI = 0E6H
00EE    00EE 362    XRI = 0EEH
00F6    00F6 363    ORI = 0F6H
00FE    00FE 364    CPI = 0FEH
00DB    00DB 365    IN = 0DBH
00D3    00D3 366    OUT = 0D3H
                                367
                                368 INSTR_SET
                                369
009E    4070 370    GOTO MVI_ENTRY
                                371
                                372 END
                                373
                                374 INSTR_DEF
                                375
```

## A-8 User Defined Assembler Code for 8080 Processor

Assembler: A8080:HP                    64000 User Definable Assembler Utility

```
376 ;*****;
377 ;   operand: reg(0-7), reg(0-7)   ;
377 ;   xxDDSSS                       +
379 ;*****;
380
0040 381   MOV = 040H
382
383   INSTR_SET
384
00A0 801B 385   CALL GET_REGISTER
00A1 050A 386   LOAD VALUE
00A2 1920 387   STORE MEM_CHECK
00A3 0183 388   SHIFT_LEFT 3
00A4 1514 389   OR OBJECT_CODE
00A5 0007 390   CHECK_COMMA
00A6 403F 391   GOTO SAVE_IO_ERROR
00A7 801B 392   CALL GET_REGISTER
00A8 150A 393   OR VALUE
00A9 C016 394   GEN_CODE ABS 8, ACCUMULATOR
00AA 2D20 395   IF MEM_CHECK <> 6 THEN GOTO CHECK_END
00AD 2D0A 396   IF VALUE = 6 THEN SAVE_ERROR IO_ERR
00B0 4015 397   GOTO CHECK_END
398
399   END

401   INSTR_DEF
402
403 ;*****;
404 ;   operand: low, high data   ;
405 ;   xxxxxxxx low, high       ;
406 ;*****;
407
0032 408   STA = 032H
003A 409   LDA = 03AH
00E2 410   JPO = 0E2H
0022 411   SHLD = 022H
002A 412   LHLD = 02AH
00C3 413   JMP = 0C3H
00DA 414   JC = 0DAH
00D2 415   JNC = 0D2H
00CA 416   JZ = 0CAH
00C2 417   JNZ = 0C2H
00FA 418   JM = 0FAH
00F2 419   JP = 0F2H
00EA 420   JPE = 0EAH
00CD 421   CALL = 0CDH
00DC 422   CC = 0DCH
00D4 423   CNC = 0D4H
```

**User Defined Assembler Code for 8080 Processor A-9**

Assembler: A8080:HP            64000 User Definable Assembler Utility

```
00CC 424        CZ = 0CCH
00C4 425        CNZ = 0C4H
00FC 426        CM = 0FCH
00F4 427        CP = 0F4H
00EC 428        CPE = 0ECH
00E4 429        CPO = 0E4H
         430
         431 INSTR_SET
         432
00B2    0000 433    EXPRESSION
00B3    2D0C 434    IF TYPE > 5 THEN SAVE_ERROR ET_ERR
00B6    C014 435    GEN_CODE ABS 8, OBJECT_CODE
00B7    E1E1 436    GEN_CODE LOW_HIGH VALUE
00B8    4015 437    GOTO CHECK_END
         438
         439 END
         440
         441
         442 INSTR_DEF
         443
         444 ;*****;
         445 ;    define storage pseudo    ;
         446 ;*****;
         447
0000    448        DS = 0
         449
         450 INSTR_SET
         451
00BA    0016 452    PRINT_LOCATION
00BB    0000 453    EXPRESSION
00BC    000D 454    CHECK_PASS1_ERROR
00BD    40C5 455    GOTO DS_ERROR
00BE    2D0C 456    IF TYPE = 0 THEN GOTO TYPE_OK
00C1    0086 457    SAVE_ERROR ET_ERR
00C2    4015 458    GOTO CHECK_END
         459 TYPE_OK
00C3    0013 460    COUNTER_UPDATE
00C4    4015 461    GOTO CHECK_END
         462
         463 DS_ERROR
00C5    008E 464    SAVE_ERROR DE_ERR
00C6    4015 465    GOTO CHECK_END
         466
         467 END
         468
         469
         470 INSTR_DEF
         471
```

## A-10 User Defined Assembler Code for 8080 Processor

Assembler: A8080:HP            64000 User Definable Assembler Utility

```

    472 ;*****;
    473 ;   define byte           ;
    474 ;*****;
    475
0000 476   DB = 0
    477
    478 INSTR_SET
    479
    480 DP_TOP
00C8 0005 481   GET_TOKEN
00C9 2D82 482   IF CLASS = 2 THEN GOTO BYTE_STRING
    483 NOT_STRING
00CC 0580 484   LOAD START
00CD 1981 485   STORE STOP
00CE 8085 486   CALL CHECK_HIGH_LOW
00CF 0000 487   EXPRESSION
00D0 2D1C 488   IF HIGH_FLAG = 1 THEN GOTO HIGH_DB
00D3 E0E2 489   GEN_CODE LOW_BYTE VALUE
00D4 40D6 490   GOTO CHECK_NEXT
    491 HIGH_DB
00D5 E0E3 492   GEN_CODE HIGH_BYTE VALUE
    493 CHECK_NEXT
00D6 0014 494   CHECK_EOL
00D7 000E 495   DONE
00D8 0007 496   CHECK_COMMA
00D9 4017 497   GOTO EOL_ENTRY
00DA 40C8 498   GOTO DB_TOP
    499
    500 BYTE_STRING
00DB 0014 501   CHECK_EOL
00DC 40E0 502   GOTO NOT_EXPR
00DD 0007 503   CHECK_COMMA
00DE 40CC 504   GOTO NOT_STRING
00DF 2981 505   DECREMENT STOP
    506 NOT_EXPR
00E0 0012 507   GET_ASCII_BYTE
00E1 40D6 508   GOTO CHECK_NEXT
00E2 114C 509   AND AND_WORD
00E3 154E 510   OR OR_WORD
00E4 C016 511   GEN_CODE ABS 8, ACCUMULATOR
00E5 40E0 512   GOTO NOT_EXPR
    513
    514 END

    516 INSTR_DEF
    517
    518 ;*****;
    519 ;   define word           ;
    520 ;*****;
    521
```

**User Defined Assembler Code for 8080 Processor A-11**

Assembler: A8080:HP            64000 User Definable Assembler Utility

```
0000 522     DW = 0
         523
         524 INSTR_SET
         525
         526 DW_TOP
00E7     0005 527     GET_TOKEN
00E8     2D82 528     IF CLASS = 2 THEN GOTO WORD_STRING
         529 NOT_STRING1
00EB     0580 530     LOAD START
00EC     1981 531     STORE STOP
00ED     0000 532     EXPRESSION
00EE     E1E1 533     GEN_CODE LOW_HIGH VALUE
         534 CHECK_NEXT1
00EF     0014 535     CHECK_EOL
00F0     000E 536     DONE
00F1     0007 537     CHECK_COMMA
00F2     4017 538     GOTO EOL_ENTRY
00F3     40E7 539     GOTO DW_TOP
         540
         541 WORD_STRING
00F4     1D1E 542     STORE_0 COUNT
00F5     0014 543     CHECK_EOL
00F6     40FA 544     GOTO NOT_EXPR1
00F7     0007 545     CHECK_COMMA
00F8     40EB 546     GOTO NOT_STRING1
00F9     2981 547     DECREMENT STOP
         548 NOT_EXPR1
00FA     0012 549     GET_ASCII_BYTE
00FB     40FF 550     GOTO DONE_STRING
00FC     C016 551     GEN_CODE ASB 8, ACCUMULATOR
00FD     251E 552     INCREMENT COUNT
00FE     40FA 553     GOTO NOT_EXPR1
         554
         555 DONE_STRING
00FF     051E 556     LOAD COUNT
0100     1001 557     AND 1
0101     2D16 558     IF ACCUMULATOR = 0 THEN GOTO CHECK_NEXT1
0104     0420 559     LOAD 20H
0105     C016 560     GEN_CODE ABS 8, ACCUMULATOR
0106     40EF 561     GOTO CHECK_NEXT1
         562
         563 END
```

End of generation, errors = 0

Words of opcodes = 568, Words of table code = 263, Total = 831

## A-12 User Defined Assembler Code for 8080 Processor

## User Defined Linker Code for 8080 Processor

---

```
FILE: L8085_Z80:I8080          HEWLETT-PACKARD: User Definable Linker

      OBJECT
LOCATION  CODE  LINE      SOURCE LINE

          1  "LINK"
          2  *****
          3  ***** 8080/85 Z80  LINKER TABLES *****
          4  *****

0000  0010      5  HEX 10   NO OF VALID CONSTANTS
0001  0002      6  HEX 0002 IDOFFSET
0002  0008      7  HEX 0008 WIDTH
0003  0008      8  HEX 0008 BASE
0004  0000      9  HEX 0000 ALIGN
0005  0005     10  HEX 0005 DIGITS0      ;#DIGITS TO DISPLAY IN PASS0
0006  0004     11  HEX 0004 DIGITS2      ;#DIGITS TO DISPLAY IN PASS2
                                ;(MAP)

0007  0000     12  HEX 0000 DBLADR
0008  0000     13  HEX 0000 SWAP
0009  0000     14  HEX 0000 IND
000A  0000     15  HEX 0000 MULTISPACE ;TRUE IFF MULTIPLE ADR SPACES
000B  FFFF     16  HEX FFFF MAXL      ;MAX LEGAL ADR ENTERABLE
                                ;IN PASS0
000C  0000     17  HEX 0000 MAXH      ;MAX LEGAL ADR ENTERBLE
                                ;IN PASS0

000D  0000     18  HEX 0000 UNDEFINED
000E  0000     19  HEX 0000 UNDEFINED
000F  0000     20  HEX 0000 UNDEFINED
0010  0000     21  HEX 0000 HISHIFT   ;SHIFT COUNT, INTERNAL TO
                                ;ACTUAL ADDRESS

0011  0000     22  HEX 0000 UNDEFINED
0012  0000     23  HEX 0000 UNDEFINED
0013  0000     24  HEX 0000 UNDEFINED
0014  0000     25  HEX 0000 UNDEFINED
0015  0000     26  HEX 0000 UNDEFINED
0016  0000     27  HEX 0000 UNDEFINED
0017  0000     28  HEX 0000 UNDEFINED
0018  0000     29  HEX 0000 UNDEFINED
0019  0000     30  HEX 0000 UNDEFINED
```

LOCATION	OBJECT CODE	LINE	SOURCE LINE
001A	0000	31	HEX 0000 UNDEFINED
001B	0000	32	HEX 0000 UNDEFINED
001C	0000	33	HEX 0000 UNDEFINED
001D	0000	34	HEX 0000 UNDEFINED
001E	0000	35	HEX 0000 UNDEFINED
001F	0000	36	HEX 0000 UNDEFINED
		37	;
		38	; LIST OF RELOCATABLE FORMATS FOR THE 8080/85 AND
		39	; Z80 ASSEMBLERS
		40	
0020	003F	41	DEF LAST-\$ ;LENGTH WORD MUST BE AT 20H
0021	0028	42	DEF FMT0 ;TWO BYTE ADDRESS, HI,LO
0022	002B	43	DEF FMT1 ;TWO BYTE ADDRESS, LO,HI
0023	002E	44	DEF FMT2 ;ONE BYTE ADDRESS, LO
			;NO RANGE CHECK
0024	0031	45	DEF FMT3 ;ONE BYTE ADDRESS, HI
			;NO RANGE CHECK
0025	0034	46	DEF FMT4 ;ONE BYTE ADDRESS, LO
			;( 0 TO 255)
0026	003B	47	DEF FMT5 ;ONE BYTE ADDRESS, LO
			;(-128 TO 127)
0027	0045	48	DEF FMT6 ;ONE BYTE P_RELATIVE
			;(-126 TO 129)
0028	0C85	50	FMT0 MOVE LOADWRD,ADR ;LOADWRD=LOADADR
0029	0056	51	LOADBYTES 2 ;LOAD 2 BYTES AND
			;LOADBYTES,,
002A	0018	52	DONE
002B	0C88	53	FMT1 SWAPBYTES LOADWRD,ADR ;LOADWRD=SWAPBYTES
			;(LOADADR)
002C	0056	54	LOADBYTES 2 ;LOAD 2 BYTES AND
			;LOADBYTES,,
002D	0018	55	DONE
002E	0C85	56	FMT2 MOVE LOADWRD,ADR ;LOADWRD=LOADADR
002F	0036	57	LOADBYTES 1 ;LOAD 1 BYTE AND
			;LOADBYTES,,
0030	0018	58	DONE
0031	0C88	59	FMT3 SWAPBYTES LOADWRD,ADR ;LOADWRD=SWAPBYTES
			;(LOADADR)
0032	0036	60	LOADBYTES 1 ;LOAD 1 BYTE AND
			;LOADBYTES,,
0033	0018	61	DONE
0034	0C85	62	FMT4 MOVE LOADWRD,ADR ;MOVE THE,ADDRESS
			;INTO LOAD WORD
0035	0036	63	LOADBYTES 1
0036	0012	64	IMMEDIATE T0 ;GET UPPER BOUND=256

## B-2 User Defined Linker Code for 8080 Processor



FILE: L8085\_Z80:I8080

HEWLETT-PACKARD: User Definable Linker

LOCATION	OBJECT CODE	LINE	SOURCE LINE
0037	0100	65	DEF 0100H
0038	00CC	66	SGE ADR,T0 ;SKIP IF ADR IS ;GE. 256
0039	0018	67	DONE
003A	0A79	68	GOTO ERROR 1 ;ADR OUT OF RANGE, ;ERROR
003B	0C85	69 FMT5	MOVE LOADWRD,ADR ;MOVE THE,ADDRESS ;INTO LOAD WORD
003C	0036	70	LOADBYTES 1
003D	0012	71	IMMEDIATE T0 ;THE UPPER 9 BITS ;SHOULD BE ALL 1'S ;OR 0'S
003E	FF80	72	DEF 0FF80H ;MASK UPPER 9 BITS
003F	0CC1	73	AND ADR,ADR,T0 ;LOOK AT UPPER 9 BITS ;OF ADR
0040	00CF	74	SNEZ ADR ;SKIP IF UPPER 9 BITS ;ARE NOT ALL 0's
0041	0018	75	DONE
0042	00CD	76	SEQ ADR,T0 ;SKIP IF UPPER 9 BITS ;ARE ALL 1'S
0043	0A79	77	GOTO ERROR1 ;ADR OUT OF RANGE
0044	0018	78	DONE
0045	0A87	79 FMT6	TWOCMP LOADWRD,LOADADR
0046	8C80	80	ADD LOADWRD,ADR,LOADWRD ;LOADWRD=ADR-LOADADR
0047	0012	81	IMMEDIATE T0
0048	FFFF	82	DEF 0FFFFH
0049	0880	83	ADD LOADWRD,LOADWRD,T0 ;LOADWRD=(ADR-LOADADR) ;-1
004A	0036	84	LOADBYTES 1
004B	0052	85	IMMEDIATE T2
004C	FF80	86	DEF 0FF80H ;GET MASK OF UPPER ;9 BITS
004D	4821	87	AND T1,LOADWRD,T2 ;T1=LOADWRD.AN.T2
004E	002F	88	SNEZ T1 ;ARE THEY ALL ZEROS?
004F	0018	89	DONE
0050	042D	90	SEQ T1,T2 ;ARE THEY ALL ONES
0051	0A79	91	GOTO ERROR1 ;UPPER 9 BITS NOT ALL ;ONES
0052	0018	92	DONE
0053	001C	93ERROR1	ERROR "Address out of range"
	144164		
005F	0018	94 LAST	DONE

ERRORS= 0

User Defined Linker Code for 8080 Processor B-3

## Note



- 
1. The first section of the linker table must contain 32 words of initialization.
  2. The next statement must be the length of the table: DEF LAST-\$.
  3. The next section is a list of addresses to formats in the linker. This list must have as many entries as formats defined in the assembler (see ASSEMBLER command section in sample program listed in Chapter 5).
  4. The label LAST must appear on the same line as the last DONE instruction.
-

## Summary of Assembler Subroutines

---

The assembler subroutines that were explained in Chapter 4 are summarized here alphabetically for quick reference.

ADD_LABEL	puts a label found in the operand field in the symbol table during pass 1. Stores VALUE and TYPE.
CHECK_AUTO_DEC	checks for auto decrement in the form of a trailing operator(s); e.g., An-.
CHECK_AUTO_INC	checks for auto increment in the form of a trailing operator(s); e.g., An+ .
CHECK_COMMA	checks the token at the STOP pointer for a comma.
CHECK_DELIMITER	checks for a delimiter at the position indicated by the STOP pointer.
CHECK_EOL	checks for a valid end of line; i.e., a blank, a semicolon, or the actual end of line.
CHECK_EXPR_ERROR	after the EXPRESSION handler is called, CHECK_EXPR_ERROR can determine if an error has been flagged by EXPRESSION.
CHECK_PASS1_ERROR	executes a return 1 when a symbolic reference is not defined in pass 1 and is defined in pass 2. Executes a return 2 if the symbolic reference is defined in both passes.

COUNTER_UPDATE	increments the program counter by the amount contained in VALUE.
ERROR code	displays an error message.
EVEN n	increments the program counter to an even word boundary if it is set to an odd value 'n' sets the program counter to the next value with 'n' trailing zeros.
EXECUTE_OPCODE	assumes that the STOP pointer is positioned at the start of a user defined opcode. It will look up the opcode, initialize OBJECT_CODE, and branch to the proper format in the user defined machine code, just as if the opcode was the first one encountered in the source statement.
EXPRESSION	evaluates expressions in the operand field and flags syntax errors in the expressions.
FIND_DELIMITER	finds the next delimiter in the present operand field.
GEN_CODE	generates absolute or relocatable object code according to the parameters chosen.
GET_ASCII_BYTE	retrieves one ASCII character from an ASCII string within quotation marks.
GET_OPCODE	checks for an opcode. Starts checking at the token indicated by the STOP pointer. Used for multiple opcodes; e.g., CMA,INA.

## C-2 Summary of Assembler Subroutines

GET_PROG_COUNTER	returns the VALUE of the user's source code program counter in the ACCUMULATOR.
GET_START_CHAR	retrieves the character indicated by the START pointer. @LABELW1 = GET_STOP_CHAR
	retrieves the character indicated by the STOP pointer. @LABELW1 = GET_SYMBOL
	checks for a symbol. Starts checking at the token indicated by the STOP pointer.
GET_TOKEN	gets the next token in the source statement. The subroutine begins at the STOP pointer and skips to the first nonblank column. A token is identified in the source statement with the START pointer at the beginning and the STOP pointer at the first column past the token.
NOT_DUPLICATE	can be used in conjunction with UPDATE_LABEL to prevent the assembler from marking a label as a duplicate.
PRINT_LOCATION	instructs the assembler to print the current VALUE of the program counter on the source listing.
SAVE_ERROR code	displays an error message.
SAVE_WARNING code	displays a warning message.



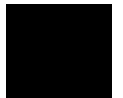
UPDATE_LABEL	allows the user to redefine the VALUE and TYPE of the label on the current line.
WARNING code	displays a warning message.



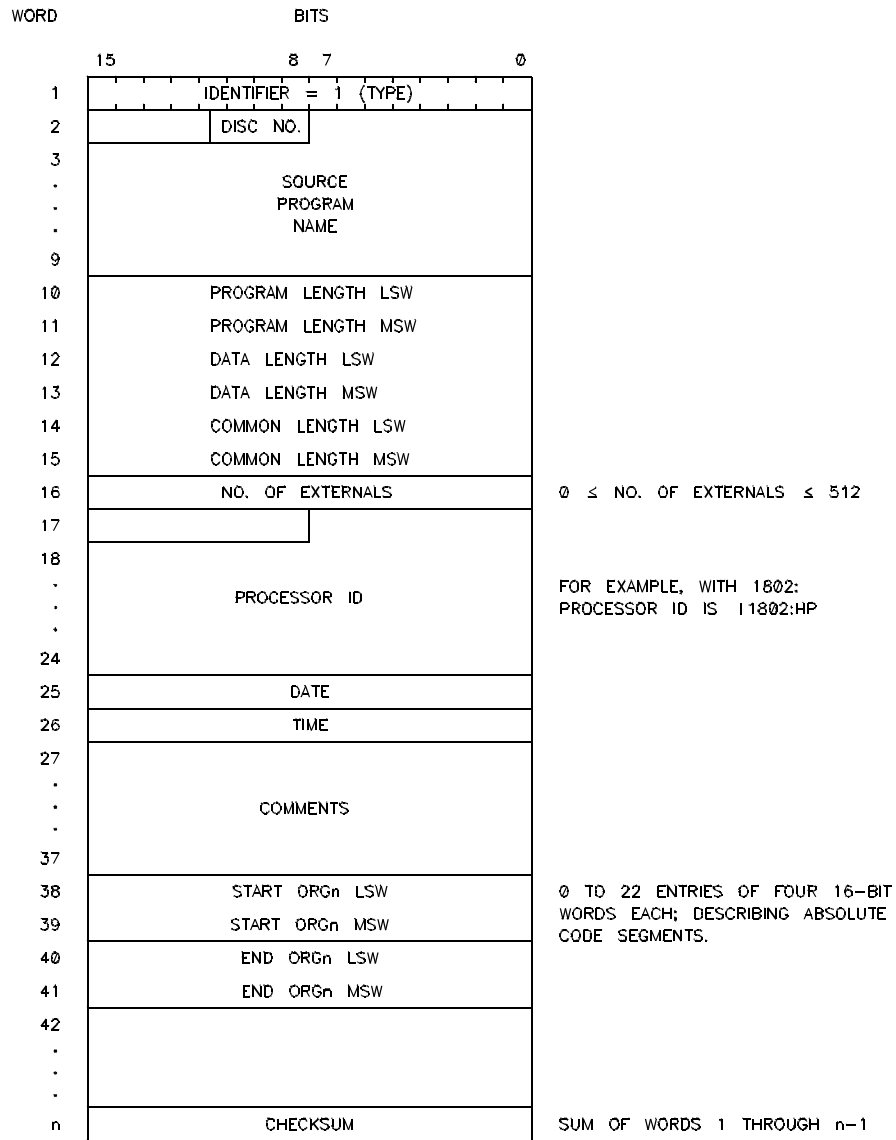
## **Relocatable and Absolute File Formats**

---

The relocatable file formats for NAM, GLB, DBL, EXT, and END records, plus the absolute file format are included here. Note that the maximum length of a record is 128 words.

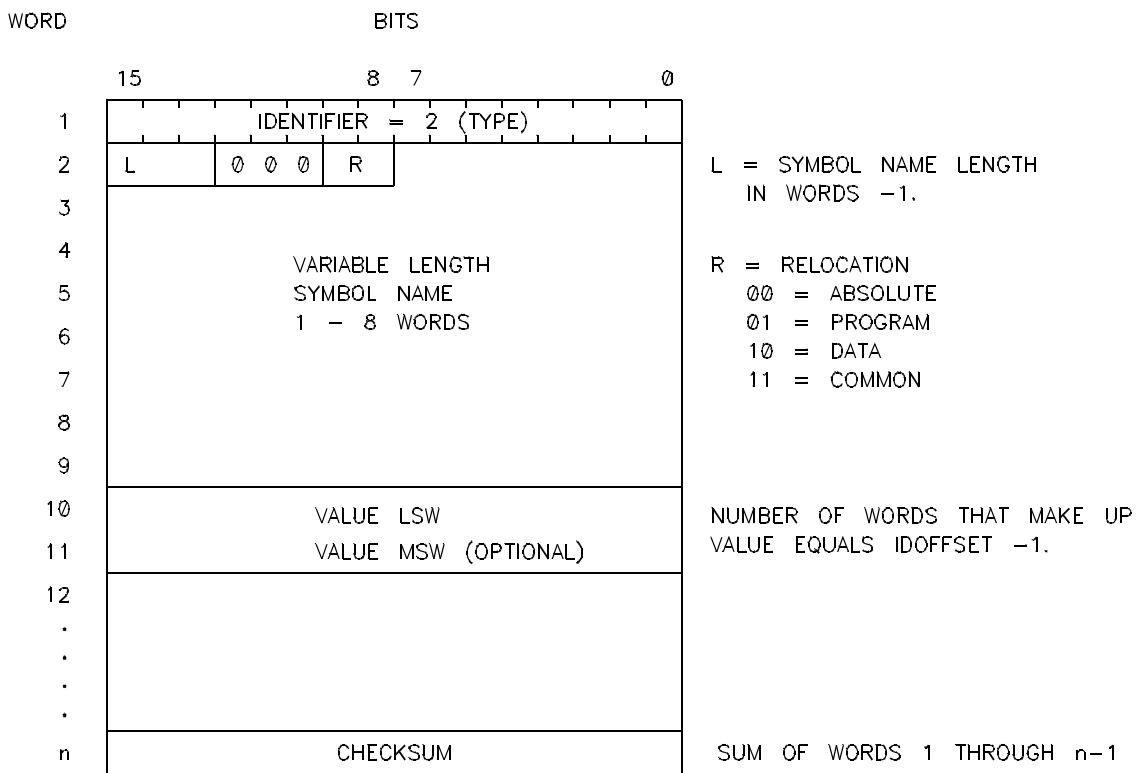


# Nam Record (record Type = 1)



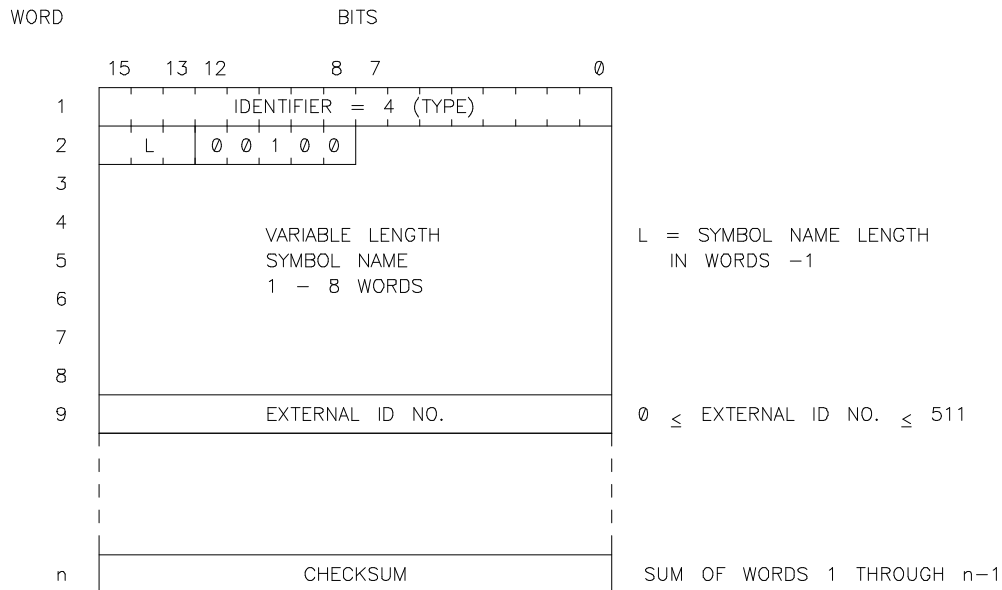


## Glb Record (record Type = 2)

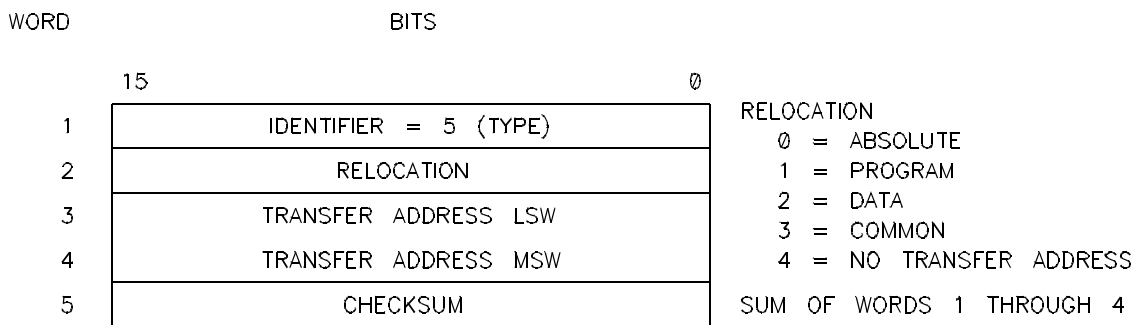




## Ext Record (record Type = 4)

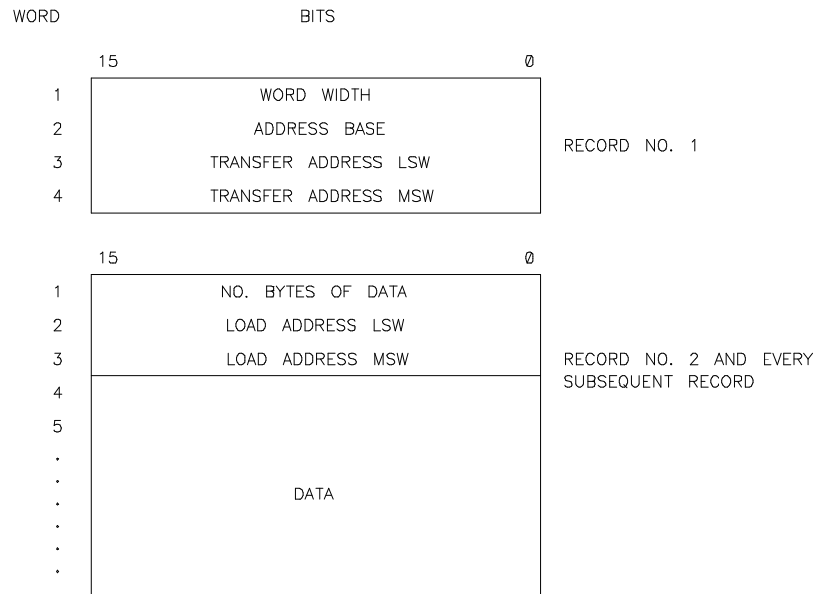


## End Record (record Type = 5)



---

## Absolute File



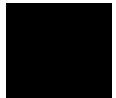
# Index

---

- A**
  - ACCUMULATOR, 3-5
  - ADD, 3-8, 7-6
  - ADD\_LABEL, 4-3
  - ADDRESS\_BASE, 3-2
  - ADR, 7-9
  - ALIGN, 7-3
  - AND, 3-8, 7-6
  - assembler directive, 3-1
  - assembler instructions, 3-8
  - assembler program, defining, 1-4
  - assembler subroutines, 4-1
  - assembler subroutines, summary, C-1
  - assembler, building process, 2-1
  - assembler, creating, 5-1
  - AUTO\_DEC\_COUNT, 3-5
  - AUTO\_INC\_COUNT, 3-5
- B**
  - BASE, 7-3
  - BLDLINK, 7-7
- C**
  - CALL, 3-8, 7-7
  - CHARACTER, 3-5
  - CHECK\_AUTO\_DEC, 4-3
  - CHECK\_AUTO\_INC, 4-3
  - CHECK\_COMMA, 4-4
  - CHECK\_DELIMITER, 4-4
  - CHECK\_EOL, 4-5
  - CHECK\_EXPR\_ERROR, 4-5
  - CHECK\_PASS1\_ERROR, 4-5
  - CLASS, 3-5
  - code formats, relocatable, 2-3
  - column pointers, 4-1
  - commands, setup, 2-2, 3-2, 5-2 - 5-3, 7-3
  - commands, setup parameters, 5-3
  - constants, internal, 2-4
  - conventions, programming, 3-11
  - COUNTER\_UPDATE, 4-7

- D** DBLADR, 7-3
- DECREMENT, 3-8
- DEF, 7-7
- delimiters, 3-11
- DIGITS0, 7-3
- DIGITS2, 7-3
- DONE, 3-8, 7-7
- DOUBLE\_ADDRESS, 3-3
  
- E** END, 3-8
- entry points, 7-6
- ERROR, 4-7, 7-7
- error messages, 4-7
- EVEN, 4-9
- EXECUTE\_OPCODE, 4-9
- EXPRESSION, 4-9
- expression types, 4-10
- EXPRESSION\_2, 4-10
- EXT\_ID\_NUMB, 3-6
- EXT\_OFFSET, 3-6
  
- F** FIND\_DELIMITER, 4-11
  
- G** GEN\_CODE, 4-11
- GEN\_CODE, absolute, 2-6
- GEN\_CODE, relocatable, 2-7
- GET\_ASCII\_BYTE, 4-12
- GET\_OPCODE, 4-12
- GET\_PROG\_COUNTER, 4-13
- GET\_START\_CHAR, 4-13
- GET\_STOP\_CHAR, 4-13
- GET\_SYMBOL, 4-14
- GET\_TOKEN, 4-15
- GOTO, 3-8, 7-7
  
- H** HISHIFT, 7-3
  
- I** IDOFFSET, 7-3
- IF...THEN, 3-8
- IMMEDIATE, 7-7
- INCREMENT, 3-9
- IND, 7-3
- INSTR\_DEF, 2-5, 5-4

- INSTR\_RESET, **3-6**
- INSTR\_SET, **5-4**
- instruction set, defining, **2-5**
  - parsing, **5-4**
- instruction set, parsing, **2-6**
- IOR, **7-7**
  
- L** LINK\_FILE, **3-2**
  - linker instructions, **7-6**
  - linker modules, **6-1**
  - linker operation, **6-1**
  - linker structure, **7-1**
  - linker, creating, **8-1**
  - LLA, **7-9**
  - LOAD, **3-9**
  - LOADADR, **7-9**
  - LOADBITS, **7-7**
  - LOADBYTES, **7-7**
  - LOADWRD, **7-10**
  - LOC\_SIZE, **3-2**
  
- M** mainframe, uploading to, **9-1**
- MAXL MAXH, **7-4**
- module, basic assembler, **1-1**
- module, basic linker, **1-1**
- module, user definable assembler, **1-1**
- module, user definable linker, **1-1**
- MOVE, **7-7**
- MULTISPACE, **7-4**
  
- N** NOP, **3-9**
- NOT\_DUPLICATE, **4-17**
  
- O** OBJECT\_CODE, **3-6**
- ONECMP, **7-8**
- OR, **3-9**
  
- P** PC\_16, **3-3**
- PRINT\_LOCATION, **4-17**
- processor, defining, **2-2, 7-4**
- PROGRAM\_COUNTER, **3-6**
- pseudo instructions, **1-3, 3-7**



pseudo numbers  
pseudo names, 3-3

**R** registers, temporary (38-40), 4-18  
RELOC\_FMT, 3-3, 7-6  
RENAME\_PSEUDO, 3-3  
RESULT, 3-6  
RETURN, 3-9, 7-8  
routines, relocatable, 7-6, 7-10

**S** SAVE\_ERROR, 4-17  
SAVE\_PTR, 3-6, 4-2  
SAVE\_WARNING, 4-17  
SCAN\_REAL, 4-17  
SEQ, 7-8  
SEQZ, 7-8  
SGE, 7-8  
SHIFT\_LEFT, 3-10  
SHIFT\_RIGHT, 3-10  
SHIFTL, 7-8  
SHIFTR, 7-8  
SIZE, 3-3  
SKELETON, 7-8  
SNEZ, 7-8  
START, 3-6  
STOP, 3-6  
STORE, 3-10  
STORE\_0, 3-10  
STORE\_1, 3-10  
SUBTRACT, 3-10  
SWAP, 7-4  
SWAPBYTES, 7-8  
SWAPWORDS, 7-8  
SYMBOLS, 3-4  
symbols, predefined, 2-4, 3-5, 7-9

**T** T0...T3, 7-10  
tables, assembler, 9-1  
TITLE, 3-4  
token classes, 4-15  
token types, 3-5  
TOKEN\_ERROR, 3-6  
TRACE, 5-6, 7-9, 8-1, 8-3



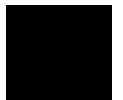
TWOCMP, 7-9  
TWOS\_COMPLEMENT, 3-10  
TYPE, 3-6  
type variables, 3-6

**U** UPDATE\_LABEL, 4-19

**V** VALUE, 3-7

**W** WARNING, 4-19, 7-7  
WIDTH, 7-4  
WORD\_SIZE, 3-4

**X** XOR, 7-9



---

## Notes

